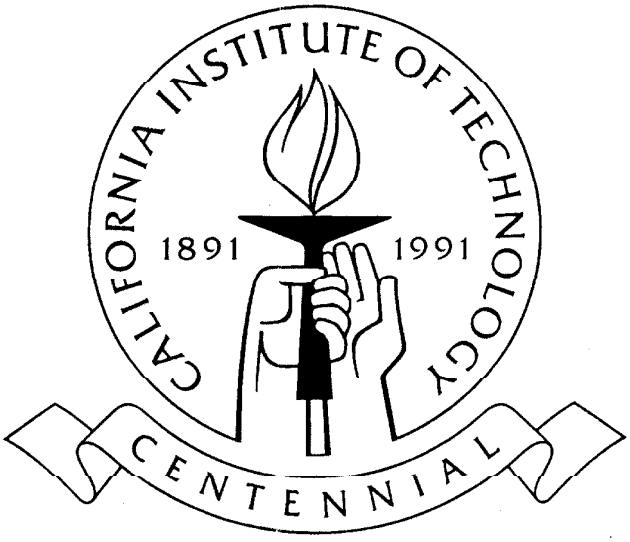

Parallel Program Design and Generalized Weakest Preconditions

Johan J. Lukkien

Computer Science Department
California Institute of Technology

Caltech-CS-TR-90-16

Parallel Program Design and Generalized Weakest Preconditions

thesis by
Johan J. Lukkien

In partial fulfillment of the requirements for the
degree of doctor of philosophy
at Groningen University, the Netherlands

California Institute of Technology
Pasadena, California

1990

(Submitted December 12, 1990)

Caltech-CS-TR-90-16

To Marike,
Klaas Jan
and Christoph

Acknowledgements

Doing research without the help of other people is unthinkable. It's no fun either. Many people have helped me during the past four years and I wish to mention some of them explicitly.

First of all I want to thank my supervisor, Jan van de Snepscheut. I have learned a lot from him during these years, especially about programming. Through him I started to appreciate formal methods in constructing programs. I thank him and Peter Hofstee for discussing with me the subjects in this thesis (and lots of other things).

I thank Peter Hilbers and Klaas Esselink for stimulating discussions on the subject of parallel computing.

I thank the members of the hobbyclub in Groningen. For me, the hobbyclub has been a source of new ideas and a place to learn from others.

I have learned from Wim Hesselink to apply the rules of a formal game without interpreting it. I thank him for his comments on the presentation of chapter five of this thesis that has improved a lot.

With respect to this fifth chapter, I also want to thank Carel S. Scholten. He gave a course on the subject of "predicate transformers" at Groningen University. The discussions in that class about the operational considerations underlying the definition of program semantics were a starting point for me to think about an operational semantics as described in chapter five.

I thank the members of the "kleine commissie", Roland Backhouse, Martin Rem and Ralph Back for carefully reading the thesis and giving me their comments.

The last year of my appointment as a "promovendus" of Groningen University I was allowed to stay at Caltech. I thank both universities for this opportunity.

Johan J. Lukkien
Pasadena, November 6, 1990

Contents

Introduction

In recent years, a lot of progress has been made in the area of VLSI design. Processors and memory have become small and cheap. It turns out that processors require much less chip area than memory. Therefore, as the overall size of the circuits decreases, it becomes feasible to equip every memory chip with a processor. Such a design leads to a machine consisting of many processors each having a small amount of private memory.

A computer consisting of more than one processor is called a *multicomputer*. A multicomputer differs essentially from conventional computers, equipped with only one processor. This is reflected in the programs that a multicomputer can execute: such a program consists of a large amount of smaller programs, one for every processor. These smaller programs are executed concurrently by the various processors and in this way the processors cooperate to perform the overall task of the machine.

In this monograph, we describe the construction of some programs for such a multicomputer and we investigate their performance. Furthermore, since these programs are more difficult to write and to understand than programs for sequential machines, we investigate methods to prove the correctness of this type of programs. In order to explain this in some detail we first have a closer look at the way multicomputers work and at the impact that this has on our programs.

There are several types of multicomputers. The type we consider in this monograph is the most general type in which every processor has its private program and its private data. Sometimes this is called a *multiple instruction, multiple data* (MIMD) machine. In order to cooperate, the processors in a multicomputer have a means to exchange information. In general there are two extreme ways to implement this. The first way is to provide for every processor, access to a global memory. The second way is that processors exchange messages with each other through communication channels or links. Machines of both types are commercially available. The first type of processor interconnection does not scale well to very large numbers of processors. The problem is that, if a large number of processors access the global memory at roughly the same time, they have to wait for each other which may slow down the machine dramatically. The second type fits the design described above of a processor per memory chip. It scales better if for every processor the number of processors with which there exists a link is limited. In this monograph, we consider the latter type of processor interconnection.

The programs for a multicomputer consist of many parts that can be executed concurrently. Each such part is called a *process*. For every processor in the multicomputer there must be at least one process in the program, but for the purpose of this introduction, we consider the simplest case in which there is exactly one process for every processor. Processes contain

instructions to communicate with other processes. Because of the physical limitations of the multicomputer (the number of links per processor is limited), every process can communicate only with a limited number of other processes, viz. the processes executed by processors to which there is a link. We come back to this restriction later.

It turns out that the multicomputer and its programs look very much the same. We may represent the multicomputer by a directed graph in which the processors play the rôle of nodes and an edge in the graph reflects the existence of a communication link from one processor to another. Similarly, we may represent the program by a directed graph in which the processes are the nodes. An edge in the graph reflects whether there will be communication from one process to another.

This leads to a program notation as introduced by C.A.R. Hoare ([17, 18]). It is known as Communicating Sequential Processes (CSP). A program written in CSP consists of a number of sequential processes. These processes can communicate using directed, point-to point channels.

In this monograph, we first study programs written in a CSP-like language. When writing programs like these, there are two major concerns: the program should be efficient (i.e., it must be possible to obtain a significant speedup when using a multicomputer) and it must be correct. In the first chapter we formalize these efficiency concerns. We show how this formalization can be used to analyze and even guide the development of two algorithms. In the second chapter we analyze a particular class of algorithms, viz. algorithms that apply a so-called broadcast. We show the correctness of the broadcast algorithm and we analyze its efficiency. In the third chapter we develop a distributed algorithm to solve an optimization problem.

As it turns out, the programs presented in those three chapters are not easy to write. During their design, explicit decisions are made that seem to be arbitrary in some cases but that greatly influence the final result. We would like to have rules and guidelines to support the development of these programs. Furthermore, the programs are complicated and it is not easy to see whether they are correct. For the smaller programs like the ones in chapters one and two the correctness can be established although the arguments involved in such a proof are often ad hoc. We were not able to give a proof of correctness for the program in chapter three. It consists of too many processes and the interaction between the processes is too complicated. In order to give such a proof we need to be able to reason about programs in a compositional way. Then we can derive properties of the program as a whole from properties of its parts, the processes.

Those concerns were the basis of the research described in the chapters four through seven. Starting point was the calculus developed for the derivation of sequential programs ([8, 9, 12]). In this calculus, the derivation of a program simultaneously provides a proof of its correctness. Correctness of a program means that it satisfies a specification. In this calculus, the specification is the condition that holds upon termination of the program. It is independent of the states reached during program executing.

For parallel programs, such a specification does not suffice. Even if the program as a whole satisfies a specification in terms of the final state, the processes in the program are, in general, not adequately specified in this way. Since the processes communicate, they may have to wait for each other at certain points. A proof of correctness has to incorporate a proof that a process does not get stuck at such a point. Furthermore, there exist many important programs that do not terminate at all (even sequential programs). These considerations caused us to look at

different ways of specifying programs.

Programs are written in some notation called the programming language. The meaning of each program, viz. its effect when executed by a computer, is described by the semantics of the programming language. In [9], the semantics of a program is described in terms of its state upon termination. This implies that no distinction is made between a property that a program has (its state upon termination) and its meaning in terms of its effect when it is executed by a machine. Introducing different specifications implies introducing different semantics of the programming language. For instance, if we want to specify that during execution of the program a certain condition becomes true, we need a different semantics than the one described in [9].

A specification may be viewed as a property that a program either has or does not have. Since we have more than one property that we want to use, we have to show that all properties that we define share a common implementation of the programming language. We therefore studied how arbitrary properties of a program may be defined on the basis of an operational semantics, i.e., a semantics which describes exactly, in terms of transitions of the machine executing the program, the meaning of the program. We generalized the notion of weakest precondition to arbitrary properties. This is described in chapter five.

In chapters six and seven we analyze two properties in great detail. One of the properties is *leads-to* as introduced by Owicki and Lamport ([32]) and later by Chandy and Misra ([6]) for their programming notation UNITY. In those two chapters we do not refer to the level of implementation anymore. This makes the analysis cleaner (only relevant aspects are taken into account) and it allows a more relaxed implementation of the programming language.

Recently, work has been published on the same topic, viz. the introduction of more general weakest preconditions. ([30, 23]). Especially the work of Morris is related to ours in that the same property is considered. The major difference however is the way sequential composition is dealt with.

The language considered in chapters five through seven is Dijkstra's guarded command language. This language more or less includes UNITY and it can be used to describe action systems ([2, 3]) hence it can be used to describe parallel programs. We reached the goal that we can specify arbitrary properties of programs written in that language and prove that a program has a certain property. We did not reach the goal that we can reason about parallel programs in a compositional way. In the last chapter we point out how parallel composition may be added to the operational semantics of chapter five.

We may ask whether programs for MIMD machines will look like CSP programs in the long run. Is it probable that this view on concurrency changes as VLSI technology makes further progress?

In the above we mentioned that processes in the program that communicate with each other are executed by processors that can communicate with each other. Some programs do not satisfy this constraint, i.e., there does not exist a mapping from the program to the multicomputer such that every process is mapped to a processor and every channel to a link. In that case we have to relax the constraint of mapping a channel to a link: it may be mapped to a path. Processes in the processors on such a path propagate messages in order to implement the channel. Such processes are called: routing processes. Thus the original program, together with the routing processes,

satisfies the constraint. In general, we may extend this to a complete routing mechanism. Then routing processes on every processor cooperate to connect every pair of processors, by passing messages through intermediate processors. The multicomputer together with the routing mechanism acts as if all pairs of processors were connected. How to create routing mechanisms is described for instance in [7, 15, 16].

Because the task that a routing mechanism performs is very clearly specified and because a routing mechanism must be fast, it is sometimes realized in hardware. In [7], the Torus Routing Chip is described which is the ancestor of a routing chip to be used in the Mosaic, an experimental machine constructed at Caltech. The Mosaic is supposed to consist of 16384 processors. From the programmers point of view this machine, together with the routing chips, looks as if it were fully connected. Since the routing chips are very fast, the memory of an arbitrary node can be accessed rather fast, hence, the combined memory of all processor acts as a global memory. However, the limitations of a global memory remain. If many processes access the memory of one processor at the same time congestion will occur, since the number of bits that can be output by a processor is the bandwidth per link times the number of links, which is limited. Therefore, the requirement on the programs remains that accesses to the combined memory be spread. Consequently, this does not change our view on concurrency. Our programs consist of processes without a global memory and communicating only with a limited number of other processes.

We end this introduction with an explanation of our notation. The programs that we are going to write are written in an extension of Dijkstra's guarded command language. This language is described in detail in [8, 9]. It has the following constructs.

abort	- loop forever
skip	- do nothing
$y := e$	- assign expression e to program variable y
$S; U$	- sequential composition
if $[](i :: B_i \rightarrow S_i)$ **fi**	- execute an S_i for which B_i holds
do $B \rightarrow S$ **od**	- repeat S as long as B holds

The **if**-statement needs some explanation. In the definition of **if** $[](i :: B_i \rightarrow S_i)$ **fi**, we have for every value of i that B_i is a boolean expression on the space in which the program variables assume their values (the state space) and S_i is a program. The range of the index i is left implicit. The meaning of **if** $[](i :: B_i \rightarrow S_i)$ **fi** is to execute one of the S_i for which the corresponding B_i holds. Execution is suspended until at least one of the B_i holds. In the absence of parallelism this is the same as waiting forever in the case that none of the B_i hold. Hence, in this case it is equivalent to *abort*, which coincides with the formal definition given in [9]. The B_i are usually called *guards*. An example is the following **if**-statement that assigns to program variable x its absolute value.

if $x \leq 0 \rightarrow x := -x$
$[]\ x \geq 0 \rightarrow$ *skip*
fi

Besides this, we have procedures and declarations of variables as in Pascal. There are some extensions to incorporate parallelism in the language, inspired by CSP ([17, 18]).

$S \parallel U$	- parallel composition
$c?y$	- input from channel c
$c!e$	- output on channel c
$\overline{c}$ (and $\overline{c!}$, $\overline{c?}$)	- the probe on channel c

We have a special type of variable called *channel*. A channel c is shared by exactly two processes. $c?y$ (input from c) in one process is synchronized with $c!e$ (output on c) in the other process. Together they implement the distributed assignment $y := e$. The *probe*, as introduced by Martin [26], is a boolean function defined on a channel c between two processes and is denoted by $\overline{c}$. $\overline{c}$ is an observation in the one process about the other process and equals the boolean value "the other process is suspended on a communication along c". We sometimes use $\overline{c?}$ or $\overline{c!}$ if we want to reason about the probes without referring to the context in which they occur. In that case $\overline{c?}$ equals the value "a process is suspended on an output on c". Similarly, $\overline{c!}$ equals "a process is suspended on an input from c".

We use **forpar** $q \in set$ **do** $s.q$ **od** to denote the parallel execution of a number of statements $s.q$, one for each value q from the set. Similarly, we use **forseq** $q \in set$ **do** $s.q$ **od** to denote the sequential execution in some arbitrary order of statements $s.q$.

If we have more than one process, we sometimes use the names of the processes as indices for variables in order to distinguish them.

When reasoning about programs, we use predicates to express conditions that may or may not hold during program execution. These predicates are boolean valued functions on the state space. Although this state space is essential for a particular program, in the general case we are not interested in it and we want it to be as anonymous as possible. We have the following two conventions. Firstly, all boolean operators are applied pointwise. If p and q are predicates on the state space, $p \wedge q$ is another predicate that is *true* in points where both p and q are *true* and *false* elsewhere. Similarly, for other operators. Secondly, universal quantification over the state space is denoted by surrounding a predicate with square brackets. Two predicates are the same if they have the same value in all points in the state space. This is denoted by $[p \equiv q]$. (Another way of denoting this is $p = q$.) On the other hand, $p \equiv q$ is a predicate that is *true* in points where p and q yield the same value and *false* elsewhere.

We often have to prove facts like $[p \equiv q]$ (p and q are the same predicate) or $[p \Rightarrow q]$ (p implies q or p is stronger than q). Suppose that we prove the latter by proving $[p \equiv x]$ and $[x \Rightarrow q]$. We give such a proof in the following format.

$$
\begin{array}{ll}
 & p \\
= & \quad \{\text{hint why } p = x, \text{ i.e., } [p \equiv x]\} \\
 & x \\
\Rightarrow & \quad \{\text{hint why } [x \Rightarrow q]\} \\
 & q
\end{array}
$$

This proof format is also used in other contexts (for instance, in proofs about properties of sets).

We use a period to denote function application. It has the highest binding power of all operators. We use the period also to denote selection of a component of an array (which

is, in fact, also function application). In order of decreasing binding power we further have $\neg, =, \wedge, \vee, \equiv, \Rightarrow$ and $\Leftarrow$.

Sometimes we refer in our proofs to the rule of Leibniz. It is sometimes referred to as "substituting equals for equals". Formally, it states that

$$p = q \;\Rightarrow\; f.p = f.q$$

for every function f of the appropriate type.

We use quantified expressions of all kinds in which we mention the bound variables explicitly. The expressions are of the form $Q(l : d.l : t.l)$ where Q is a quantor (such as $\forall, \bigcup$), l is a list of bound variables, $d.l$ is the domain expression and $t.l$ the term of the quantification. Q must be an associative commutative operator with a zero element. The quantification over an empty domain, i.e., $d.l = \mathit{false}$, is defined to be equivalent to this zero element. Examples of quantified expressions are

$$\forall(r : r \in \mathbb{R}^+ : \exists(s : s \in \mathbb{R} : r \cdot s = 1)),$$
$$\bigcup(i : i \in I : A_i).$$

The last expression is traditionally written as

$$\bigcup_{i \in I} A_i,$$

in which the bound variable is implicit. We have one quantified expression that we write a little differently, viz. the set constructor.

$$\{l : d.l : t.l\}$$

is equivalent to

$$\bigcup(l : d.l : \{t.l\}).$$

The proof format that is used here is described in detail in [11, 9].

Chapter 1

The performance of parallel programs

1.1 Introduction

An important reason to employ a huge number of processors to solve a particular problem is that, in this way, the problem may be solved faster. Given a parallel program, it would be nice if it were possible to express this increase in speed. In the literature, two measures are used for this, the *speedup* and the *efficiency* ([27]). The speedup compares the execution time of the program when it runs on a network of processors with the single processor case. The efficiency is a measure for the utilization of every processor in the network.

Usually, speedup and efficiency are not defined very precisely. In this chapter we give a precise definition. We define them for a particular program and network of processors, as functions of the input complexity of the algorithm and the size of the network. Because the definition is precise, we can do some arithmetical manipulation with the formulae and derive some simple, elementary results. We relate this with the rule of thumb known as "Amdahl's law" and we use it to develop and analyze two algorithms.

1.2 Formalization

We consider an execution of a program, S, on a network, G, of processors. The network may be thought of as a directed graph as described in the introduction. The number of nodes (= processors) of the network is denoted by p. An execution of S depends on the specific input given to that execution. The complexity of this input is denoted by n. We will not decorate our formulae with S and G although they will depend on both but give the respective definitions as functions of p and n. This means that in fact we investigate worst case behavior with respect to all networks of size p and all inputs of size n. The time required for the execution is denoted by $T.p.n$ and we assume it to be positive and finite. This time is spent by all processors, in computing, in communicating or in being idle. This is expressed in the following definition.

Definition 1 *For processor i,*

$b_i.p.n$ = *the time processor i spends computing,*
$c_i.p.n$ = *the time processor i spends communicating,*
$y_i.p.n$ = *the time processor i is idle.*

□

Since every processor spends $T.p.n$ it follows

$$\forall(i :: b_i.p.n + c_i.p.n + y_i.p.n = T.p.n). \tag{1.1}$$

Obviously, b, c and y are nonnegative. In order to avoid large expressions, we introduce shorthands for the sums.

$$B.p.n = \Sigma(i :: b_i.p.n)$$
$$C.p.n = \Sigma(i :: c_i.p.n)$$
$$Y.p.n = \Sigma(i :: y_i.p.n)$$

Now we have

$$B.p.n + C.p.n + Y.p.n = p\,T.p.n. \tag{1.2}$$

Any program written in our programming language is built from basic statements (assignments and communication statements). Such a basic statement cannot be distributed any further. Hence, every program containing at least one assignment statement cannot be executed any faster than the execution of this assignment statement takes. A program may be written in such a way that not one but a number of assignments are put together to be executed sequentially. Therefore, we associate with any program S a *granularity* $g.n$, which represents the execution time that is obtained by taking the minimum of the execution times of these sequential parts of S. At least one processor has to spend this time, hence

$$\exists(i :: b_i.p.n \geq g.n) \tag{1.3}$$

independently of G. Furthermore, no processor can execute a substantial part of the algorithm any faster than $g.n$, hence

$$\forall(i :: b_i.p.n \geq g.n \vee b_i.p.n = 0). \tag{1.4}$$

This granularity can be different for two algorithms solving the same problem but it is positive. For instance, when in a matrix multiplication algorithm the inner product is defined to be a basic operation, $g.n = (2n-1)f$, where f is the time required to perform one floating point operation. Another algorithm may have a smaller granularity but at some point, the multiplication of two floating point numbers has to be done.

Next we want to express the speedup gained when 2 or more processors are used to execute S. The performance of S on 2 or more processors is then compared with the single processor case. Notice that we are investigating S and not an arbitrary program that solves the same problem. Hence, S might be an algorithm with a very good speedup function but also be a very bad program to solve the problem. In view of this we define the speedup as follows.

Definition 2

$$s.p.n \;=\; \frac{B.1.n}{T.p.n}$$

□

We use $B.1.n$ here rather than $B.p.n$ because this is more honest. In general we may expect B to be a non-decreasing function in both arguments. In many cases, B does not depend on p, for instance, when S is an algorithm that performs, for every n, a fixed number of operations and spreads those operations over the available processors. In such a case no operation will be duplicated due to the parallelism.

Property 3 $B.1.n \leq B.p.n \;\Rightarrow\; s.p.n \leq p$

Proof.

$$\begin{array}{ll} & \dfrac{B.1.n}{T.p.n} \leq p \\ \Leftarrow & \quad \{B.1.n \leq B.p.n, C.p.n \geq 0, Y.p.n \geq 0\} \\ & \dfrac{B.p.n + C.p.n + Y.p.n}{T.p.n} \leq p \\ = & \quad \{(1.2)\} \\ & \text{true} \end{array}$$

□

Property 4

$$\begin{array}{l} B.1.n \leq B.p.n \\ \Rightarrow \\ (s.p.n = p) \;\equiv\; \forall(i :: b_i.p.n = \dfrac{B.1.n}{p} \wedge c_i.p.n = 0 \wedge y_i.p.n = 0) \end{array}$$

Proof.

$$\begin{array}{ll} & s.p.n = p \\ = & \quad \{\text{definition}\} \\ & pT.p.n = B.1.n \\ = & \quad \{(1.2)\} \\ & B.p.n + C.p.n + Y.p.n = B.1.n \\ = & \quad \{B.1.n \leq B.p.n, C.p.n \geq 0, Y.p.n \geq 0\} \\ & C.p.n = 0 \wedge Y.p.n = 0 \wedge B.p.n = B.1.n \\ = & \quad \{(1.1), \text{definitions } B, C \text{ and } Y\} \\ & \forall(i :: b_i.p.n = \dfrac{B.1.n}{p} \wedge c_i.p.n = 0 \wedge y_i.p.n = 0) \end{array}$$

□

These two properties reflect that, in the usual case that the computation time increases with p, the speedup is bounded by p. Furthermore, optimal speedup is possible if and only if every processor performs an equal amount of the computation and does not communicate with other processors.

From (1.1) and (1.3) it follows that $T.p.n \geq g.n$ hence,

Property 5 $s.p.n \leq \frac{B.1.n}{g.n}$.

□

Next we investigate how the speedup function behaves when n and p become large. We assume $B.1.n = B.p.n$, i.e., the amount of work for the parallel algorithm is the same when it is executed on 1 processor as when it is executed on p processors. First we concentrate on the dependence of s on p. We show

$$\underline{\mathrm{N}}(i :: b_i.p.n = 0) > (p - \frac{B.1.n}{g.n}) \tag{1.5}$$

by contraposition.

$$\begin{array}{ll} & \underline{\mathrm{N}}(i :: b_i.p.n = 0) \leq (p - \frac{B.1.n}{g.n}) \\ \Rightarrow & \quad \{(1.4)\} \\ & \underline{\mathrm{N}}(i :: b_i.p.n \geq g.n) > p - (p - \frac{B.1.n}{g.n}) \\ = & \quad \{\text{arithmetic}\} \\ & \underline{\mathrm{N}}(i :: b_i.p.n \geq g.n) > \frac{B.1.n}{g.n} \\ \Rightarrow & \quad \{\text{definition } B.p.n\} \\ & B.p.n > \frac{B.1.n}{g.n} g.n \\ = & \quad \{\text{arithmetic}\} \\ & B.p.n > B.1.n \end{array}$$

This contradicts our assumption $B.1.n = B.p.n$. This means that employing more than $\frac{B.1.n}{g.n}$ processors results in processors contributing nothing to the computation part. For the communication part of the work we have a similar restriction as for the computation part: a basic communication statement cannot be divided. It follows that if p becomes very large, the speedup does not increase anymore. Since the speedup has a bound that is independent of p (property 5) there exists an optimal number of processors to execute S with a given input size.

If we fix n and vary p we evaluate the performance of S with fixed input on varying networks. In many cases, this is the only analysis that is done (see the next section). This must lead to disappointing results since, due to the existence of the granularity upper bound, it is not possible to employ an arbitrarily large number of processors to solve a fixed problem.

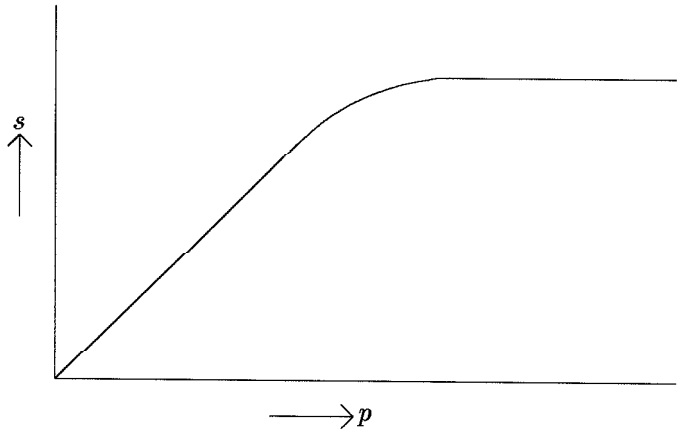

Figure 1.1: Possible speedup function

Next consider p to be fixed. We use (1.2) to rewrite $s.p.n$

$$s.p.n = \frac{p}{1 + \frac{C.p.n + Y.p.n}{B.1.n}}$$

We may assume B and C to be increasing functions of n. If the overhead term increases faster than $B.1.n$, we find an optimal speedup for some n_0. (This n_0 might be 1 if the overhead term is big, right from the start.) If we increase n beyond n_0, the speedup can drop even below 1, which means that we had better not used more than one processor.

If $B.1.n$ increases in the same way as the overhead term, s tends to some limit: $\frac{p}{1+a}$; if $B.1.n$ increases faster than the overhead terms, s tends to p.

What we get out of this is that problem size and network size need to be tuned. In general there will be an optimal number of processors to execute an algorithm with a fixed input complexity. Larger networks require larger problems in order to use the processors efficiently.

In general, it is not very wise to look for an optimal speedup for a fixed problem size. Suppose that the speedup function has a shape as in figure 1.1. As can be read from the figure, we obtain almost the optimal speedup with only half the number of processors as required for the optimal case. We had therefore better the *increase with p* in speedup as a measure. We then look for the minimal p that satisfies $s.(p+1).n - s.p.n < c$, for a constant $c, 0 \leq c \leq 1$. When we choose $c = 0$, this is the same as maximizing $s.p.n$.

Sometimes another performance measure is suggested, the *efficiency* function. It is defined by

Definition 6

$$e.p.n = \frac{s.p.n}{p}.$$

□

$e.p.n$ is a measure of processor utilization. The efficiency function corresponding to the speedup function of figure 1.1 is given in figure 1.2. The restriction might be posed that the efficiency

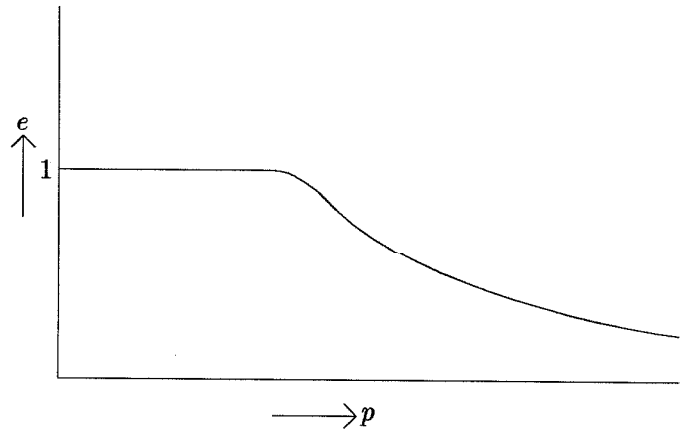

Figure 1.2: Efficiency function corresponding to figure 1.1

be greater than, for instance, 0.50. Under the assumption that B does not decrease with p, the efficiency function is most probably a descending function of p. This can be seen as follows.

$$
\begin{array}{ll}
 & e.(p+1).n < e.p.n \\
= & \quad \{\text{definitions of } s \text{ and } e\} \\
 & \dfrac{B.1.n}{(p+1)T.(p+1).n} < \dfrac{B.1.n}{pT.p.n} \\
= & \quad \{(1.2)\} \\
 & B.p.n + Y.p.n + C.p.n < B.(p+1).n + Y.(p+1).n + C.(p+1).n \\
\Leftarrow & \quad \{B.p.n \leq B.(p+1).n\} \\
 & Y.p.n + C.p.n < Y.(p+1).n + C.(p+1).n
\end{array}
$$

Hence, under the above assumption, the efficiency is decreasing if the overhead term increases with p, which is very likely to be the case. Therefore, looking for an optimal efficiency does not make much sense.

1.3 Amdahl's law

Amdahl's law is a rule used to estimate the number of processors that can be utilized to execute a certain algorithm. In the general formulation of Amdahl's law, the computation time is split into a serial part ($x.n$) and a parallel part ($z.n$). The serial part needs to be done sequentially; the parallel part can be done in parallel on a network of processors. This gives us

$$B.1.n = x.n + z.n.$$

Only the parallel part of the computation can be executed by the network. If we assume that all parallelism can be exploited we have

$$T.p.n = x.n + \frac{z.n}{p}.$$

By using definition 2 we obtain

$$\begin{aligned} s.p.n &= \frac{x.n + z.n}{x.n + \frac{z.n}{p}} \\ &\leq 1 + \frac{z.n}{x.n}. \end{aligned}$$

This relation is known as Amdahl's law. The ratio $\frac{z.n}{B.1.n}$ is sometimes called the *degree of parallelism*. Usually, the dependence on n is omitted. Indeed, if there is no dependence on n, i.e., the serial portion of the work is a constant fraction of the overall work, there is a fixed limit on the speedup. For instance, if the serial part is $\frac{B.1.n}{10}$, the speedup is bounded by 10. This observation can also be derived from property 5. However, in many useful algorithms there is a dependency on n and this upper bound can become arbitrarily large if n increases. This has also been noticed in [33].

1.4 Matrix multiplication

In this section we show how the formalization of the second section can help us to develop and analyze an algorithm. We are given two square matrices P and Q of dimension n and we are interested in a program to compute the product matrix $R = P \cdot Q$. We consider a distributed implementation of the "traditional" algorithm. For the computation of one element of R, denoted by $R.i.j$, n multiplications and $n-1$ additions are needed: each $R.i.j$ is the sum of n terms, each term being a product $(P.i.k)(Q.k.j)$ for some $k, 0 \leq k < n$. This means that $B.1.n = n^2(2n-1)f$ (where f is the time required to perform one floating point operation). The summation can be done in $\lceil \log n \rceil$ steps using distributed recursive doubling. In principle, the products can be computed in one step if enough processors and enough copies of P and Q are available. This yields for the fastest algorithm, $g.n = (\lceil \log n \rceil + 1)f$, for $p \geq n^3$. This gives a lower bound for $T.n^3.n$ of $(\lceil \log n \rceil + 1)f$ which might be achieved for instance on a fully connected network of size n^3.

This lower bound is indeed a lower bound: in general we are not in the position to scale our network with the problem size; furthermore, it is not obvious that this lower bound can be achieved with a limited amount of point-to-point connections (in this example it is possible with, for instance, a mesh connected tree as the interconnection network).

We are interested in an analysis in which network size and problem size are dealt with independently. Therefore, we analyze the problem from the point of view of data distribution. The elements of P, Q and R must reside in some processor and we assume that they are distributed more or less evenly among the available processors. Initially, only one copy of P and Q is available. We also limit the number of processors such that every processor contains at least one element of all three matrices. For the purpose of the analysis, we assume that p is a divisor of n. Hence, every processor contains $\frac{n^2}{p}$ elements of all three matrices. We spread the amount of work evenly among the processors, giving $b_i.p.n = \frac{n^2(2n-1)f}{p}$ for $0 \leq i < p$. This means that per element of the result matrix, $\frac{n^2(2n-1)}{p} / \frac{n^2}{p} = 2n-1$ floating point operations have to be performed, which, of course, corresponds to one inner product. We adopt the following restriction.

Each processor performs all multiplications and additions necessary to compute the part of matrix R that resides in that processor.

Next we compute how the matrices should be distributed. We consider the communication term for a fixed processor, i say. The time necessary to transmit one item (matrix element) over one link is denoted by d. For an element $R.j.k$ that resides in processor i, row j of P and column k of Q are needed in order to compute $R.j.k$. We try to distribute R in such a way that as few elements of P and Q are needed as possible. We look at the different row- and column indices of the elements of R in i. Define

$$x = \underline{\mathrm{N}}(j : 0 \leq j < n : \exists(k : 0 \leq k < n : R.k.j \text{ resides in } i)),$$
$$y = \underline{\mathrm{N}}(j : 0 \leq j < n : \exists(k : 0 \leq k < n : R.j.k \text{ resides in } i)).$$

x represents the number of different column indices of the elements of R in i and y the number of different row indices. x columns of P and y rows of Q are needed to compute all elements of R in i. This yields at least $n(x+y)$ communications. Furthermore, since $\frac{n^2}{p}$ elements of R reside in i, $xy \geq \frac{n^2}{p}$. Minimizing $n(x+y)$ with respect to this constraint yields $x = y = \frac{n}{\sqrt{p}}$. If we assume that p is a square and that it is possible to use the parts of P and Q that already reside in i, we obtain the following formula as a lower bound for the communication term.

$$c_i.p.n = (\frac{2n^2}{\sqrt{p}} - \frac{2n^2}{p})d$$

In a particular algorithm this term will be a bit worse since we only counted the "incoming" communications of a processor and forgot about the "outgoing" communications. We will see this when we derive an algorithm that approaches this lower bound. In principle, the lower bound for the idle time is zero. However, it is unclear whether we can approach this.

We now turn to the development of an algorithm. In the preceding part it was derived that a block-like (or more precisely: grid-like) partitioning is optimal with respect to the communication overhead. We give an algorithm that uses this data partitioning and which has a torus as the underlying network. The torus has $p = q^2$ nodes named (i,j) for $0 \leq i,j < q$. There is a link from (i,j) to both $(i+1,j)$ and $(i,j+1)$ where addition is modulo q. The matrices are distributed as follows: processor (i,j) contains the elements of P, Q and R with indices k and l such that $iq \leq k < (i+1)q$ and $jq \leq l < (j+1)q$. We treat these parts of the matrices as square submatrices, denoted by $P[i,j], Q[i,j]$ and $R[i,j]$ respectively. Now we can write a postcondition for the algorithm in processor (i,j).

$$R[i,j] = \Sigma(k : 0 \leq k < q : P[i,k] \cdot Q[k,j])$$

Here, multiplication and addition refer to operations on square submatrices. An invariant for a repetition is

$$R[i,j] = \Sigma(k : 0 \leq k < l : P[i,k] \cdot Q[k,j]) \wedge 0 \leq l \leq q$$

giving the following program for processor (i,j).

$$
\begin{array}{l}
l := 0; R[i,j] := 0; \\
\textbf{do}\ l \neq q \rightarrow R[i,j] := R[i,j] + P[i,l] \cdot Q[l,j]; \\
\qquad\qquad l := l+1 \\
\textbf{od}
\end{array}
$$

A disadvantage of this program is that in step l of the computation, $P[i,l]$ is needed in each processor in row i of the network and $Q[l,j]$ in each processor in column j. This might result in an increase in idle time, since these parts need to be broadcasted through the network. With a different choice of the invariant, we can change the algorithm in such a way that information can be obtained from neighboring processors only. Instead of the index l we use a linear function of i,j and l. Addition and subtraction for the indices is performed modulo q. The following invariant does the job.

$$R[i,j] = \Sigma(k : 0 \leq k < l : P[i,k+i+j] \cdot Q[k+i+j,j]) \wedge 0 \leq l \leq q$$

This results in the following program.

$$
\begin{array}{l}
l := 0; R[i,j] := 0; \\
\textbf{do}\ l \neq q \rightarrow R[i,j] := R[i,j] + P[i,l+i+j] \cdot Q[l+i+j,j]; \\
\qquad\qquad l := l+1 \\
\textbf{od}
\end{array}
$$

The parts of P and Q that are needed in (i,j) in step l can be obtained from a neighboring processor which used the same parts in the previous step of the computation. We now add variables and channels for the message passing. We use channels $h.k.l, 0 \leq k,l < q$ for the message passing in the horizontal direction and $v.k.l, 0 \leq k,l < q$ for the message passing in the vertical direction mapped directly to the links of the torus. The values of the variables are defined in the following invariant.

$$s = P[i,l+i+j] \ \wedge \ t = Q[l+i+j,j] \tag{1.6}$$

In order to establish this invariant, we need two repetitions. In the first one we establish $s = P[i,i+j]$ and in the second one $t = Q[i+j,j]$. Invariant for the first repetition is $s = P[i,k+j] \wedge 0 \leq k \leq i$. We obtain the following program.

$$s := P[i,j]; k := 0; \textbf{do}\ k \neq i \rightarrow stmp := s;\ (h.i.j?s \parallel h.(i+1).j!stmp); k := k+1\ \textbf{od}$$

We have used here the temporary variable *stmp* to accommodate the simultaneous sending of an old value and the receiving of a new value. For t we have a similar initialization. In order to maintain the invariance of (1.6) we have to perform communications as well. The complete program is given below.

$$
\begin{array}{l}
s := P[i,j]; k := 0;\ \textbf{do}\ k \neq i \rightarrow stmp := s;\ (h.i.j?s \parallel h.(i+1).j!stmp); k := k+1\ \textbf{od}; \\
t := Q[i,j]; k := 0;\ \textbf{do}\ k \neq j \rightarrow ttmp := t;\ (v.i.j?t \parallel v.i.(j+1)!ttmp); k := k+1\ \ \textbf{od}; \\
l := 0; R[i,j] := 0; \\
\textbf{do}\ l \neq q \rightarrow R[i,j] := R[i,j] + s \cdot t; \\
\qquad\qquad stmp := s; ttmp := t; \\
\qquad\qquad h.i.j?s \parallel h.(i+1).j!stmp \parallel v.i.j?t \parallel v.i.(j+1)!ttmp; \\
\qquad\qquad l := l+1 \\
\textbf{od}
\end{array}
$$

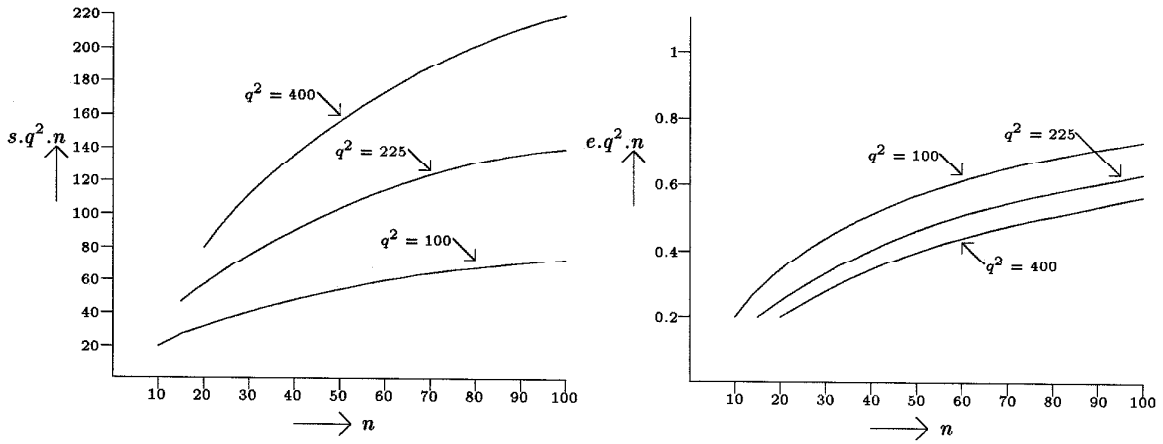

Figure 1.3: Speedup and efficiency as functions of n

The communication time for the local algorithms can be computed by counting the number of communications performed. In the initialization at most $4(q-1)$ communications take place; in the body exactly $4q$. Therefore,

$$y_i.q^2.n + c_i.q^2.n = \frac{(8q-4)n^2}{q^2}d, \quad 0 \leq i < q^2.$$

If we assume that there is no further idle time involved, this gives us the total time spent by the algorithm.

$$T.q^2.n = \frac{2n^2(2n-1)f + (8q-4)n^2d}{q^2}$$

This gives us the following speedup and efficiency functions.

$$s.q^2.n = \frac{2q^2(n-1)f}{2(n-1)f + (8q-4)d}$$
$$e.q^2.n = \frac{2(n-1)f}{2(n-1)f + (8q-4)d}$$

Figure 1.3 shows speedup and efficiency as functions of n for various values of q^2. We have assumed that $d = f$, i.e., that the time required to perform one floating point operation is the same as the time required to communicate a matrixelement over one link. As we see both speedup and efficiency are increasing functions of n. If n becomes very large compared to q^2, the speedup becomes optimal and so does the efficiency. For one particular value of n we can increase the speedup by adding more processors at the expense of using the processors less efficiently. This can also be seen in figure 1.4 which gives speedup and efficieny as functions of q^2 for various values of n. The speedup increases as more processors are used but the efficiency decreases.

We conclude with some remarks about the algorithm.

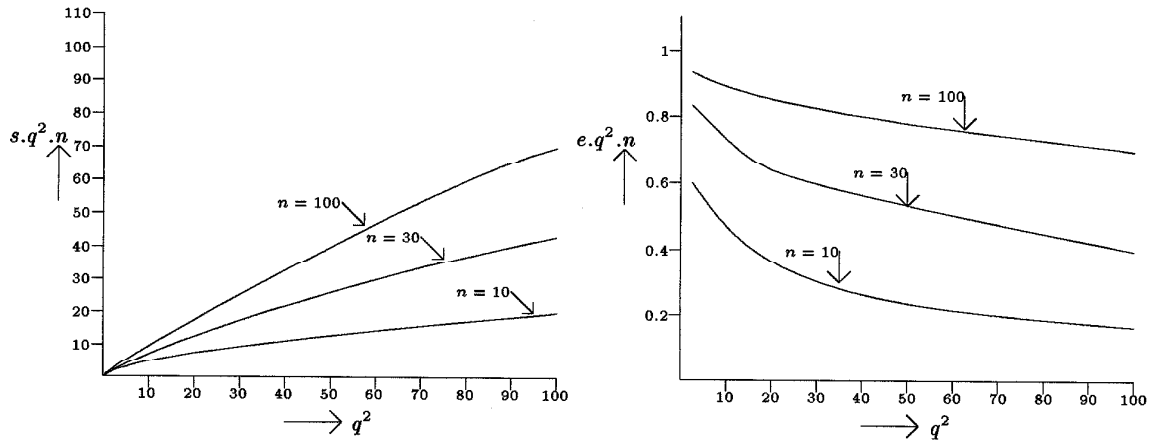

Figure 1.4: Speedup and efficiency as functions of q^2

- For any algorithm that spreads the work evenly and that restricts the number of communications to be $O(n^2)$, the speedup will converge to p if n becomes very large since the computation time is $O(n^3)$. However, the algorithm presented here also performs well if $q = n$.

- In general, p is not a divisor of n. This implies a somewhat different block-partitioning which gives slightly different formulae.

- In the above algorithm, not all possible parallelism is exploited. The communications can be done in parallel with the computation if the hardware allows it, reducing the communication term dramatically. One can even think of performing a "look ahead" of one block.

- In counting the number of communications performed, we counted all communications. However, as can be seen in the above program, many of them are done in parallel.

- If the algorithm is run on a different network using a routing mechanism, d is not a constant but a function of the network, especially of the network size.

1.5 Load balancing of a processor farm

In the following example it is not our aim to derive an algorithm but to investigate a particular algorithm from the point of view of *load balancing*. When more than one processor is used to execute an algorithm, the amount of work has to be spread evenly among the processors. The problem of how to do so is known as the problem of load balancing. A distinction can be made between *static* load balancing and *dynamic* load balancing. In the former case the amount of work per processor is fixed and known in advance. In the latter case, balancing the load is part of the algorithm: during execution, tasks are moved from one processor to another.

With the formalization of section two we can define a perfect load balance.

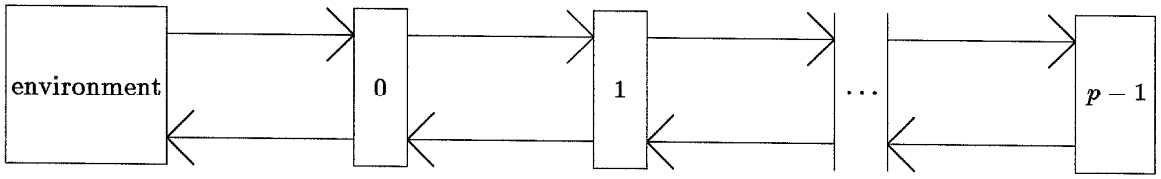

Figure 1.5: Line topology

Definition 7 *Algorithm S on network G with input complexity n has a perfect load balance if*

$$\forall(i,j :: b_i.p.n + c_i.p.n = b_j.p.n + c_j.p.n).$$

□

Notice that this is equivalent to the statement that all processors have the same amount of idle time. Notice also that a perfect load balance does not imply minimal execution time. However, it follows from property 4 that a perfect load balance is a necessary condition for maximal speedup (in the case that the computation time does not decrease with p).

We analyze a particular algorithm from the point of view of load balancing. Given are two sets V and W and a function $f : V \to W$. The problem is to compute $f.V$ on a network of processors: the elements of V are fed into the network from outside (the environment), and the elements of $f.V$ are returned by the network. A straightforward algorithm is: let every processor have its private part of V, V_i say, and compute $f.V_i$. This algorithm is known as a processor farm ([28]). Every processor i has three tasks: receive and forward elements of V (for other processors), receive and forward elements of $f.V$ and compute f for elements of V_i. We do not give the program text here. We concentrate on load balancing in the case of two topologies: a line and a pipeline. In the case of a line topology, input and output to the network is done at the same processor (figure 1.5); for a pipeline, input is done at the one end, while output is done at the other end of the pipeline. The latter topology allows more parallelism (figure 1.6).

If the topology of the network is a line, processor i is connected to processors $i-1$ and $i+1$ with an exception for 0 and $p-1$. $p-1$ has no successor and 0 has the environment as its predecessor. V is fed into 0 and $f.V$ is retrieved from 0. First we introduce some names. We assume that the computation of f takes the same time r for all elements of V. Because of this assumption, only the sizes of the local parts of V, V_i, matter. We denote these by n_i. $|V|$ is denoted by n. Communication of an element of V over one link takes v time; communication of an element of W w time. Now we have

$$b_i.p.n = rn_i, \qquad B.p.n = rn.$$

Because of the line topology, processor i has to forward elements of V for all processors with a higher number than i. If an element of V passes through i it has to be input and output by i hence it counts twice. This gives us

$$c_i.p.n = (v+w)n_i + 2(v+w)\Sigma(j : i < j < p : n_i).$$

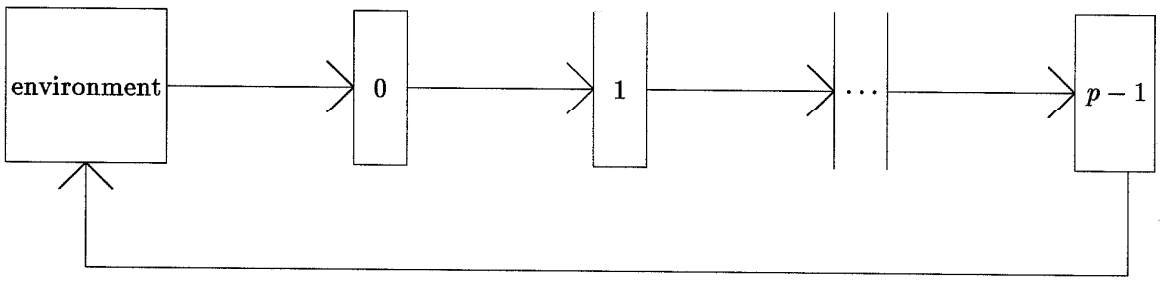

Figure 1.6: Pipeline topology

We want to derive a closed expression for n_i assuming that a perfect load balance is achieved. We assume n_i to be positive. Consider two neighboring processors, i and $i+1$. From the perfect load balance we obtain

$$
\begin{aligned}
& b_i.p.n + c_i.p.n \;=\; b_{i+1}.p.n + c_{i+1}.p.n \\
= \quad & \{\text{definition}\} \\
& rn_i - rn_{i+1} + (v+w)n_i + 2(v+w)n_{i+1} - (v+w)n_{i+1} = 0 \\
= \quad & \\
& (v+w+r)n_i + (v+w-r)n_{i+1} = 0 \\
= \quad & \\
& n_{i+1} = \frac{r+v+w}{r-v-w} n_i .
\end{aligned}
$$

Since n_i and n_{i+1} are supposed to be positive, we have $r > v + w$. Now let $\frac{r+v+w}{r-v-w}$ be denoted by K. Notice that $K > 1$. We have $n_i = K^i n_0$. Furthermore,

$$
\begin{aligned}
& \Sigma(j : 0 \le j < p : n_j) = n \\
= \quad & \{\text{substitution}\} \\
& \Sigma(j : 0 \le j < p : K^j n_0) = n \\
= \quad & \{\text{calculus}\} \\
& \frac{K^p - 1}{K - 1} n_0 = n \\
= \quad & \\
& n_0 = \frac{K-1}{K^p - 1} n .
\end{aligned}
$$

We can now write a closed expression for the load of processor 0.

$$
\begin{aligned}
& b_0.p.n + c_0.p.n \\
= \quad & \\
& rn_0 + (v+w)n_0 + 2(v+w)\Sigma(j : 0 < j < p : n_j)
\end{aligned}
$$

$$=$$

$$rn_0 + 2(v+w)n - (v+w)n_0$$

$$=$$

$$2(v+w)n + (r-v-w)\frac{K-1}{K^p-1}n$$

We have two limit cases: $r \downarrow v+w$ (the computing time hardly exceeds the communication time) and $r >> v+w$ (the computing time is much bigger than the communication time). In the first case the expression tends to $2n(v+w)$. This reflects the time required to feed V into a single processor, compute $f.V$ and retrieve $f.V$. In the second case the expression tends to $2n(v+w) + \frac{r}{p}n$. This reflects an almost optimal speedup. Notice that, with the above assumptions of a line topology and $r > v+w$, the term $2n(v+w)$ is a lower bound for $T.p.n$.

Next we perform a similar computation for a pipeline topology: elements of V are fed in at 0 and elements of $f.V$ are retrieved at $p-1$. We then have

$$c_i.p.n = 2w\Sigma(j : 0 \leq j < i : n_j) + 2v\Sigma(j : i < j < p : n_j) + n_i w + n_i v.$$

From the definition of perfect load balance we obtain in the same way as above

$$n_{i+1} = \frac{r+v-w}{r+w-v}n_i.$$

From n_i positive, it follows $r > |v-w|$. Let M be defined as $\frac{r+v-w}{r+w-v}$. It follows that M is positive. We distinguish between the two cases: $w > v$ and $w \leq v$. In the first case, $r > w-v$ and in the second case $r > v-w$. The closed expression for processor 0 becomes

$$b_0.p.n + c_0.p.n \;=\; 2nv + n(r+w-v)\frac{M-1}{M^p-1}.$$

If M tends to 1 this expression tends to $2nv + n\frac{r+w-v}{p}$ which, as a function of p has a better lower bound than the line case. If $w > v$ and $r \downarrow w-v$, the expression tends to $2nw$ (since $M \to 0$). If $w \leq v$ and $r \downarrow v-w$, the expression tends to $2nv$ (if also $p > 1$). We see that we have the maximum of $2nv$ and $2nw$ as a lower bound when the computation time becomes comparable with the communication time. This is better than the lower bound in the line case which was the same as the single processor case. It reflects that in a pipeline the output can be done in parallel with the input. However, we may not assume to have a low idle time because the output of $f.V$ can start only after the elements have passed the whole pipeline. This latency increases with p.

Chapter 2

Topology independent algorithms based on spanning trees[1]

2.1 Introduction

In this chapter, we consider a class of programs that can be run on every network without adding a routing mechanism. The programs that we consider are characterized by the fact that all communications to be performed by the processors executing the algorithm are between one processor on the one hand and every other processor on the other hand. Either the one processor may transmit information to all other processors, often called a *broadcast*, or information from all processors may be combined into a single datum such as in the case of recursive doubling. We show how such algorithms may be executed on an arbitrary network when only local information about the structure of the network is available in each processor.

2.2 Definitions and notations

As before, a network of processors is represented by a graph with p nodes. We restrict our attention to undirected, connected networks. (The algorithms presented here can be extended to directed, strongly connected networks at the expense of additional notation and additional storage.) For fixed r, we consider a spanning tree of the network rooted in processor r. The tree is represented by array *father* as follows: for processor i, $i \neq r$, $father_i.r$ is the father of processor i in the tree with root r. Similarly, we define *sons*: for each processor i, $sons_i.r$ is the set of sons of processor i in the tree with root r. We have the following relationship:

$$j \in sons_i.r \;=\; (j \neq r \wedge i = father_j.r).$$

Parameter r allows us to consider a number of spanning trees at the same time: one for every processor. All these trees are used in the algorithms below. The local information about the network structure stored in each processor i consists of $sons_i.r$ and $father_i.r$, for each r. Hence, only part of the tables is available locally, viz. how the direct neighbors are arranged in each

[1]This chapter has been published in: Beauty is our business, a birthday salute to Edsger W. Dijkstra, Texts and monographs in computer science (D. Gries, series editor), Springer-Verlag, New York, 1990.

spanning tree. Notice that we are free to choose the spanning trees to be the ones that have shortest paths to the root. A (distributed) algorithm to compute such spanning trees can be found in [5].

In the programs to follow, we abuse the program notation a little bit. We use *father* as if it were a channel, and *sons* as if it were a set of channels. Furthermore, we use these channels to communicate values in both directions. This makes the presentation more clear since extra names for channels are avoided. In an implementation, this has to be taken care of.

2.3 Some examples

We give some examples to illustrate the use of the tables. Assume that processor r contains a value v that must be transmitted to all processors in the network. Processor i has local variable x_i for this purpose. In other words, the broadcast should establish $\forall(i :: x_i = v)$. For each processor i the following program does the job.

if $i = r \rightarrow x_i := v \; [] \; i \neq r \rightarrow father_i.r?x_i$ **fi**;
forpar $j \in sons_i.r$ **do** $j!x_i$ **od**

The correctness of the program follows by induction on the structure of the spanning tree with root r. For each processor i, the first statement assigns v to x_i and the second statement assigns v to x_j for every son j of i.

In the second example each processor i has local variable x_i and the goal is to compute the maximum of all x_i in processor r. Using variables m_i and tmp_i per processor i, the program is

$m_i := x_i$;
forpar $j \in sons_i.r$ **do** $j?tmp_i$; $m_i := m_i$ **max** tmp_i **od**;
if $i = r \rightarrow skip \; [] \; i \neq r \rightarrow father_i.r!m_i$ **fi**.

Again the correctness follows by induction. The value output by processor i is the maximum of all x_i in the subtree of which i is the root. This maximum is computed by selecting the maximum of the local variable (first line) and the respective maxima of its subtrees (second line).

The next example concerns the multiplication of two square matrices. For the sake of simplicity, we assume that the dimension of the matrices equals p (the number of processors). The problem is to compute $C = A \times B$ where A and B are given $p \times p$ matrices in which rows and columns are numbered 0 through $p-1$. Initially, processor i contains row i of A and column i of B, denoted by $A.i.*$ and $B.*.i$ respectively. After the computation, processor i contains row i of C. The algorithm is described in two steps. The abstract version consists of p inner product steps per processor.

$$j := 0; \textbf{do } j \neq p \rightarrow C.i.j := (A.i.*) \cdot (B.*.j); j := j + 1 \textbf{ od}$$

In the refined version we take into account that $B.*.j$ is not stored in processor i but in j. During step j each processor requires a copy of column j of B and we insert the broadcast algorithm for distributing it. The above algorithm contains a "hidden" synchronization in the form of the shared variable j. We replace j by a private variable j_i per processor. This leads to the following algorithm.

$j_i := 0;$
do $j_i \neq p \rightarrow$ **if** $i = j_i \rightarrow b_i := B.*.j_i$ [] $i \neq j_i \rightarrow father_i.j_i?b_i$ **fi**;
forpar $k \in sons_i.j_i$ **do** $k!b_i$ **od**;
$C.i.j_i := (A.i.*) \cdot b_i;$
$j_i := j_i + 1$
od

The correctness of this matrix multiplier is not at all obvious. Because the various j_i's may have different values, different processors may be engaged in the broadcasting phase using different spanning trees. In the following section we show that, nevertheless, the above algorithm is free from deadlock and computes the correct result.

Our final example concerns the transitive closure of a directed graph. Nodes in the graph are numbered 0 through $p-1$ and the set of edges is given as a boolean matrix $B.i.j$. The transitive closure can be computed (in situ) by Warshall's algorithm ([36]).

$k := 0;$
do $k \neq p \rightarrow i := 0;$
do $i \neq p \rightarrow j := 0$
do $j \neq p \rightarrow B.i.j := B.i.j \vee (B.i.k \wedge B.k.j);$
$j := j + 1$
od;
$i := i + 1$
od;
$k := k + 1$
od

The outer loop maintains the invariant

$B.i.j =$ there is a path from i to j via nodes with a number smaller than k.

This expression is abbreviated to $B.i.j = i \xrightarrow{k} j$. The correctness of the algorithm follows from the above invariant and from the property that for all i, $k \xrightarrow{k} i = k \xrightarrow{k+1} i$ and $i \xrightarrow{k} k = i \xrightarrow{k+1} k$. This property implies that it does not matter whether the "old" or the "new" values of $B.*.k$ and $B.k.*$ are used. This again implies that, for every k, the p^2 assignments can be done in any order or in parallel.

For the parallel algorithm, we have a similar distribution as in the previous example. Processor i contains row i of B, denoted by $B.i.*$. The p iterations of the middle repetition will be done in parallel. This yields p executions of the following algorithm.

$k := 0;$
do $k \neq p \rightarrow j := 0;$
do $j \neq p \rightarrow B.i.j := B.i.j \vee (B.i.k \wedge B.k.j);$
$j := j + 1$
od;
$k := k + 1$
od

In the assignment, only $B.k.j$ is not available on processor i. We insert the broadcast for establishing $b_i = B.k.*$ for local array b_i on i. Again we introduce local variables k_i and j_i for processor i.

$k_i := 0;$
do $k_i \neq p \rightarrow$ **if** $i = k_i \rightarrow b_i := B.k_i.* \ [] \ i \neq k_i \rightarrow father_i.k_i?b_i$ **fi**;
forpar $q \in sons_i.k_i$ **do** $q!b_i$ **od**;
$j_i := 0;$
do $j_i \neq p \rightarrow B.i.j_i := B.i.j_i \vee (B.i.k_i \wedge b_i.j_i);$
$j_i := j_i + 1;$
od;
$k_i := k_i + 1$
od

This algorithm has the same communication behavior as the previous one and therefore its proof of correctness relies on the same argument.

2.4 Proof of correctness

In this section we prove the correctness of the matrix multiplication algorithm. To be correct, the assertion $b_i = B.*.j_i$ must hold before the assignment to $C.i.j_i$. This is the case if $j_i = i$, but if $j_i \neq i$ it must be the case that the correct value was communicated to i by $father_{j_i}$, and that value is the value broadcast by the root of spanning tree j_i.

The code for broadcasting a value to all nodes of a spanning tree from the root of the spanning tree was already given and proved correct in section three (first example), and this code is what appears in the multiplication program. However, since the same code is used to broadcast values for n different spanning trees, we have to show that two communicating processors work on the same spanning tree.

For processors i and k, and for integer j, we define $\{i,k\}$ $\underline{\text{in}}$ j as the number of times that the link between i and k (if any) occurs in the spanning trees, rooted in 0 through $j-1$. Formally, for all i, k and j such that $0 \leq i < p,\ 0 \leq k < p,\ 0 \leq j \leq p$,

$$\{i,k\} \ \underline{\text{in}} \ j = \underline{\text{N}}(h : 0 \leq h < j : k \in sons_i.h \vee i \in sons_k.h).$$

Notice that the integer $\{i,k\}$ $\underline{\text{in}}$ j is an ascending function of j. We need one more piece of notation. For processors i and k, $|i,k|$ is the number of completed communications between i and k. It increases every now and then as the execution of the program proceeds.

Next, we formulate an invariant of the program. We claim that

$$\{i,k\} \ \underline{\text{in}} \ j_i = |i,k| \tag{2.1}$$

holds, for all i and k, at the beginning of each iteration of the loop. The claim holds initially because $j_i = 0$ initially, which implies $\{i,k\}$ $\underline{\text{in}}$ $j_i = 0$, and because no communications have been completed initially, which implies $|i,k| = 0$. In each iteration of the loop j_i is increased by 1. Hence, for those k that are neighbors of i in the tree rooted in j_i the left-hand side increases by 1 on account of the definition of $\underline{\text{in}}$, and the right-hand side increases by 1 because

a communication between i and k is performed. For those k that are not neighbors of i in the tree rooted in j_i, both sides remain unchanged.

From the fact that (2.1) holds at the beginning of each iteration of the loop, and from the fact that j_i is incremented at the end, we conclude that $\{i,k\}$ $\underline{\text{in}}$ $j_i \leq |i,k|$ holds at any moment. By symmetry, it follows that $\{i,k\}$ $\underline{\text{in}}$ $j_k \leq |i,k|$ holds. From the invariant and from the definition of $\underline{\text{in}}$ we also derive that, in processor i, the precondition of a communication with k is:

j_i is the maximum value of j for which $\{i,k\}$ $\underline{\text{in}}$ $j = |i,k|$.

Together with our observation that

j_k is a value of j for which $\{i,k\}$ $\underline{\text{in}}$ $j \leq |i,k|$

the monotonicity of $\underline{\text{in}}$ in j implies that $j_i \geq j_k$ as a precondition of a communication with k. This property is the quintessence of the correctness argument.

Since $j_i \geq j_k$ holds in i as a precondition of the communication between i and k, and $j_k \geq j_i$ holds as a precondition of the same communication in k, $j_i = j_k$ holds when the communication is performed. This implies that i and k are involved in a broadcast on one and the same spanning tree, an algorithm of which we have indicated the correctness before. As a result, the correctness of the matrix multiplication algorithm follows.

We show that no deadlock occurs. If deadlock occurs during execution of a program then a cycle of waiting processors exists: i is suspended on a communication with k, k on another one, etc., and a certain processor on i. Since $j_i \geq j_k$ if i waits on k, it follows that all processors in the cycle have the same value for their respective j's, which implies that all links in the cycle are from the same spanning tree: a contradiction. Hence, no deadlock occurs.

2.5 Performance

We investigate the overhead of the broadcast as used for instance in the matrix multiplier of section three. We choose a very simple example: assume that processor r contains an array v of size k. We repeat the program from section three for a broadcast that establishes $\forall(i :: b_i = v)$ for local array b_i on processor i.

if $i = r \rightarrow b_i := v$ [] $i \neq r \rightarrow father_i.r?b_i$ **fi**;
forpar $j \in sons_i.r$ **do** $j!b_i$ **od**

We are interested in the behavior of the broadcast algorithm on a fixed network, i.e., we investigate the performance as a function of k. The time required to finish the broadcast is denoted by $t.k$, i.e., the time that elapses between initiation of the program by root r and its completion by all processors. (It corresponds to the overhead term $c_i.p.k + y_i.p.k$.) We are interested in upper and lower bounds for $t.k$.

In order to give an accurate estimate for $t.k$ we need to formalize some characteristics of the network. The time required to communicate one item (element of the array) along one link is denoted by d. The **forpar** construct starts the parallel execution of several statements. There is a penalty to be paid for such a construct since there will be some memory management and process management associated with it and, possibly, physical i/o has to be started. We denote

the time required for this by s (notice that in general s will depend on the number of statements that is started by **forpar**). There is a third parameter, c, the maximum depth of any spanning tree. In the case of shortest path spanning trees, e equals the diameter of the network.

A lower bound for $t.k$ can be derived from the program text. If $e > 1$, at least one processor performs k inputs and k outputs. (There are possibly more outputs but since they are performed in parallel only one counts.) The startup time s only counts for the output. An upper bound for $t.k$ can be found by looking at a leaf in the tree. The distance to the root is at most e. Only after having received the entire message, a process communicates the message to its sons in the tree. Communicating the entire message to a neighbor in the tree takes $s + kd$ time units. This gives the following formula

$$s + 2kd \leq t.k \leq e(s + kd). \tag{2.2}$$

This scheme corresponds to store-and-forward routing as described for instance in [22].

There is another way to perform the same broadcast. Instead of communicating the entire message to a neighbor, a processor can receive one item, forward it, receive the next and so on. This corresponds to "wormhole" routing ([7]). The program text for this way of message transmission is given by the following fragment.

$j_i := 0;$
do $j_i \neq p \rightarrow$ **if** $i = r \rightarrow b_i.j_i := v.j_i$ [] $i \neq r \rightarrow father_i.r?b_i.j_i$ **fi**;
forpar $q \in sons_i.r$ **do** $q!b_i.j_i$ **od**;
$j_i := j_i + 1$
od

Again we derive a lower bound from the program text. If $e > 2$ at least one processor must both receive and transmit the array. This takes $k(d + s + d)$ time units. Again an upper bound can be found by looking at a leaf in the tree. For a leaf with distance e to the root the input of the first item completes after $e(s + d)$ units. The transmission of the next $k - 1$ elements takes $(k - 1)(s + 2d)$ units, which is the iteration time of the predecessor of this leaf. In order to avoid confusion we denote the time required for wormhole routing by $w.k$. This gives us the formula

$$k(s + 2d) \leq w.k \leq e(s + d) + (k - 1)(s + 2d). \tag{2.3}$$

Notice that in this case the difference between the upper bound and the lower bound is small (it does not depend on k). Notice also that it can be done more efficiently by performing the output of one item in parallel with the input of the next. This would drop the factor 2 in (2.3). However, if no routing hardware is available, s is most likely to dominate d.

The upper bounds in the above two formulae give the time to finish the entire operation. For (2.3) this will be a good prediction but for (2.2) not necessarily. In store-and-forward routing only neighboring processors have to synchronize. An algorithm that uses a broadcast a number of times might therefore perform better if this way of routing is applied, even if c is large.

We have done some experiments to demonstrate this point. We have run the matrix multiplier on a ring network of 17 INMOS Transputers. The constants s and d have been measured. For s, it was not possible to measure an accurate value because it strongly depends on the current state of the machine (e.g. the amount of available memory).

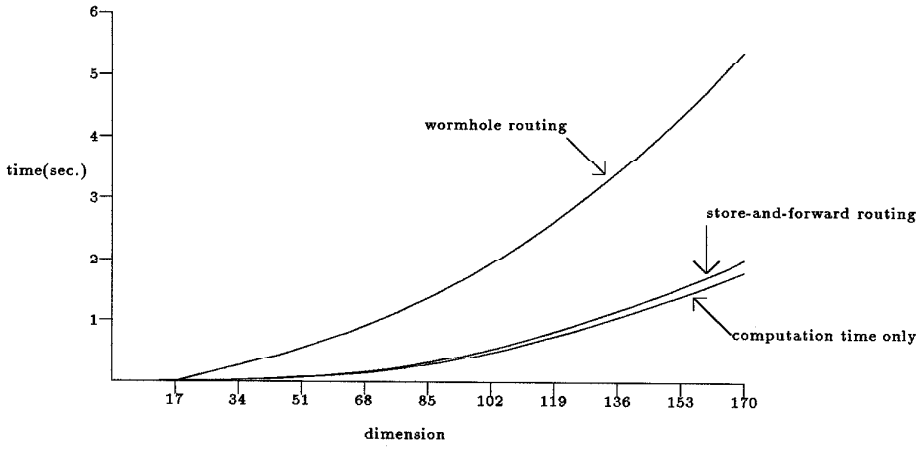

Figure 2.1: results of experiments

$d = 2.25 \cdot 10^{-6}$ sec.
$5.0 \cdot 10^{-5} \leq s \leq 1.0 \cdot 10^{-4}$ sec.
$e = 8$

The matrix multiplier has been run with and without communications for varying dimensions of the matrix. The results are given in figure 2.1. The communication time for store-and-forward routing equals the lower bound of (2.2). The performance of wormhole routing is rather bad due to the high value of s. This routing technique in fact requires dedicated hardware to reduce s drastically, in which case it performs much better ([7]).

2.6 Final remarks

The method described here is applicable to a relatively small class of algorithms. But for this class it is efficient since information is routed through the network using shortest paths and each link will contain a certain message at most once. If point-to-point routing is used, information transmitted from one processor to every other processor is duplicated from the start ($p-1$ copies will be transmitted).

All the information about the topology is contained in the tables. The algorithms presented here do not depend on the topology anymore.

As VLSI technology progresses, it will be possible to have several processors on one chip. The connections among these processors will be much faster than connections with processors on another chip. The method presented here offers the possibility to use these fast links by weighting the links when the shortest path tables are computed.

The experimental results show that the time complexity of the method is not always worst case. If used in a regular way, it may be a good choice to apply a broadcast.

Chapter 3

A parallel algorithm for a class of optimization problems

3.1 Introduction

We consider a class of optimization problems which is described informally as follows. Given is a rooted tree $T = (X, E)$ where X is the set of nodes and E the set of edges, and a cost function $c : X \to \mathbb{R}$ non-decreasing on every path from the root to any leaf. The problem is to determine a leaf which has minimal cost.

In general the tree T is very big and it is not feasible to enumerate the nodes. Thus the whole tree is not given, only a method to construct it. A problem then is to prune the tree that is generated as soon as possible using the monotonicity property of c. We first define the problem formally.

Definition 8 *Let f be a function from a set X to the powerset of X, $\mathcal{P}(X)$.*

$$\begin{aligned} f^0.x &= \{x\} \\ f^{i+1}.x &= \bigcup(y : y \in f^i.x : f.y) \text{ for } i \geq 0 \\ f^*.x &= \bigcup(i : i \geq 0 : f^i.x) \end{aligned}$$

□

In the following, quantifications range over X unless explicitly stated otherwise. We can state the problem as follows.

Definition 9 *Optimization problem.*

Given:
A finite set of nodes, X,
An element $r \in X$, the root,
A successor function $s : X \to \mathcal{P}(X)$ for which
$\forall(x, y : x \neq y : s.x \cap s.y = \emptyset) \ \wedge \ \forall(x :: r \notin s.x) \ \wedge \ X = s^*.r$,
A cost function $c : X \to \mathbb{R}$ for which $\forall(x, y : y \in s.x : c.y \geq c.x)$.

Determine m such that
$s.m = \emptyset \ \wedge \ \forall(x : s.x = \emptyset : c.x \geq c.m)$
(i.e., a leaf with minimal cost).

□

The condition "$X = s^*.r$" has been added to restrict our attention to the interesting part of the nodeset viz. the descendants of r. Notice that this definition indeed covers a large class of problems. Suppose for instance that nothing is known about the value of the cost function in the internal nodes, i.e., an exhaustive search through all the nodes is necessary. Then c can be defined to be zero in the internal nodes and the definition covers this problem too. In that case however, the following algorithm may not be the best one to choose.

There are three methods that are used to solve this kind of problems: depth-first search, breadth-first search and best-first search or branch and bound (see [20] for a discussion of the latter). A disadvantage of the last two methods is that they may use an exponential amount of memory. We therefore exploit only the first method in which we use the following obvious property[1].

Property 10 *For nodes n and m,*
$c.n \geq c.m \ \Rightarrow \ \forall(x : x \in s^*.n : c.x \geq c.m)$.

□

We first derive a sequential algorithm. Then we show which aspects lead to the introduction of parallelism. In a number of steps we modify the sequential algorithm into a parallel one. A very important design consideration is that the parallel algorithm should dynamically balance the load on the processors executing the algorithm.

3.2 A sequential algorithm

We are given a global variable, m, which invariantly is a leaf of the tree. We want to write a procedure p that, given a node n establishes

$\{m = M \ \wedge \ s.m = \emptyset\}$
$p(n)$
$\{s.m = \emptyset \ \wedge \ c.M \geq c.m \ \wedge \ \forall(x : x \in s^*.n \wedge s.x = \emptyset : c.x \geq c.m)\}$.

Thus m may be assigned a leaf with minimal cost in $s^*.n$ if this leaf is a better solution than m, otherwise m may retain its value.

In order to determine for an arbitrary node n the new value of m, we must check the successors of n. In some cases we can conclude directly that no better solutions will be found from property 10. In all other cases we have to perform the same algorithm recursively. The correctness of the algorithm depends on the associativity and commutativity of the operator **min**.

[1]Sometimes not only best-first search but any algorithm that uses this property is called a branch and bound algorithm. The author did not succeed in finding a clear definition in the literature.

procedure p (n: node);
$\{m = M \wedge s.m = \emptyset\}$
begin if $s.n = \emptyset \rightarrow$ **if** $c.n < c.m \rightarrow m := n$
[] $c.n \geq c.m \rightarrow$ *skip*
fi
[] $s.n \neq \emptyset \rightarrow$ **if** $c.n < c.m \rightarrow$ **forseq** $k \in s.n$ **do** $p(k)$ **od**
[] $c.n \geq c.m \rightarrow$ *skip*
fi
fi
end
$\{s.m = \emptyset \ \wedge \ c.M \geq c.m \ \wedge \ \forall(x : x \in s^*.n \wedge s.x = \emptyset : c.x \geq c.m)\}$

m is assigned a correct value by a call of $p(r)$ if the initial value of m is an arbitrary leaf in the tree. This can be established by a simple initial computation like

$m := r$; **do** $s.m \neq \emptyset \rightarrow m :=$ an arbitrary element of $s.m$ **od**.

This computation is guaranteed to terminate because X is finite and the graph represented by X and s is acyclic.

The condition $c.n < c.m$ guarding the recursive call of p can be replaced by *true*. It does not affect the correctness of the algorithm, only the efficiency. Thus it does not matter whether an assignment to m is delayed provided m is never assigned a value with higher cost than it had. We will use this in the parallel algorithm.

3.3 A parallel algorithm

We want to construct a parallel algorithm, i.e., we want to solve the problem using several communicating processes. These processes run on several processors. During the execution of the algorithm, a processor should receive work from its neighboring processors when it runs out of work. We choose a very simple interconnection pattern for the processors: a directed ring with size M and develop a program in which the interconnection pattern of the processes can be mapped directly on this ring. In the following, the first index of a process indicates the number of the processor on which it runs. Thus a process with index i, can input from processes with index $i - 1$ and output to processes with index $i + 1$, arithmetic being modulo M.

From the final remark of the previous section we conclude that it is possible to distribute the global variable m. On each processor i, a process $collect_i$ maintains a local copy of m. If $collect_i$ receives a leaf with lower cost than the leaf it already has, it should communicate this to its successor in the ring. In this way, if at some time some *collect* process has a leaf with minimal cost, eventually all *collect* processes will have a leaf with this minimal cost. To avoid deadlock in the situation that every *collect* process tries to communicate with its successor, we add a one place buffer between every two *collect* processes. Now we can give the text for the processes as far as this communication in the ring is concerned. In $collect_i$, the boolean w indicates whether information should be transmitted to a successor. The channels $ctb.i, 0 \leq i < M$ ("*collect* to *buffer*") and $btc.i, 0 \leq i < M$ are used to connect the processes (see figure 3.1).

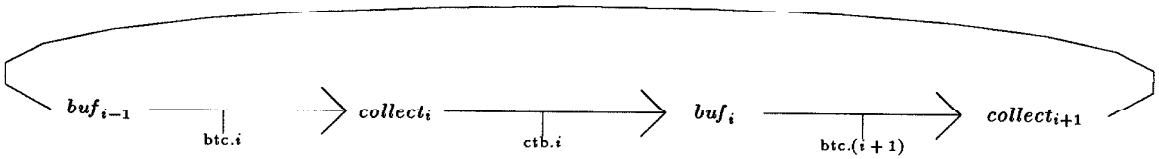

Figure 3.1: Ring connection of *collect* processes and buffers

buf_i :: [**var** l: node;
do $true \rightarrow ctb.i?l;\ btc.(i+1)!l$ **od**
]

$collect_i$ :: [**var** m, l: node; w: bool;
m := "arbitrary leaf"; $w := true$;
do $true \rightarrow$ **if** $\overline{btc.i} \rightarrow btc.i?l;$
if $c.l < c.m \rightarrow m := l;\ w := true$
[] $c.l \geq c.m \rightarrow skip$
fi
[] $w \wedge \overline{ctb.i} \rightarrow ctb.i!m;\ w := false$
fi
od
]

Besides these *collect* processes on every processor, we need processes like the procedure p above that perform the computation on the tree. These processes must be able to communicate with the ring of *collect* processes in order to read the current value of $c.m$ and to assign a new value to m if a better one is found. Furthermore, it must be easy to take some work from such a process and transfer it to another processor. Therefore we make the recursive calls in p more explicit: each incarnation of p is replaced by a process. This implies that we will have an array of processes similar to p; instead of calling p recursively, each process communicates the parameter to its successor in the array. These processes are called p_{ij}, where i is the processor number and j is the number in the process array. We will assume that we know the maximal depth of the tree, N say. First we rewrite the procedure p as described above. Then we eliminate the comparisons with m and modify the *collect* processes so that the communications with p_{ij} are taken into account. We employ a square of channels, $ptp.i.j, 0 \leq i < M \wedge 0 \leq j < N$ ("p to p") for the communications among these processes p_{ij} and another such square called ptc ("p to *collect*") for the communications between the p processes and the *collect* processes.

```
p_ij :: [ var n: node;
          do true → ptp.i.j?n;
                    if s.n = ∅ → if c.n < c.m → ptc.i.j!n
                                 [] c.n ≥ c.m → skip
                                 fi
                    [] s.n ≠ ∅ → if c.n < c.m → forseq k ∈ s.n do ptp.i.(j+1)!k od
                                 [] c.n ≥ c.m → skip
                                 fi
                    fi
          od
        ]
```

Because m is not available to p_{ij} we cannot compute the boolean value "$c.n < c.m$" directly. We introduce an extra variable, tm, in p_{ij} which is an estimate for $c.m$: $tm \geq c.m$. Since $c.n \geq tm \Rightarrow c.n \geq c.m$, in many cases a comparison with tm will be enough. If $c.n < c.m$, tm must be updated by asking the current value of $collect_i$. We use the array ctp ("*collect* to p") for these communications.

```
p_ij :: [ var n: node; tm: real;
          tm := ∞;
          do true → ptp.i.j?n;
                    if s.n = ∅ → if c.n < tm → ptc.i.j!n; tm := c.n
                                 [] c.n ≥ tm → skip
                                 fi
                    [] s.n ≠ ∅ → if c.n < tm → ctp.i.j?tm
                                 [] c.n ≥ tm → skip
                                 fi;
                                 if c.n < tm → forseq k ∈ s.n do ptp.i.(j+1)!k od
                                 [] c.n ≥ tm → skip
                                 fi
                    fi
          od
        ]
```

Now we have to modify the *collect* processes in order to incorporate the communications with p_{ij}. Since p_{ij} commits unconditionally to a communication, $collect_i$ can simply wait for such a communication. There is no potential for deadlock due to cyclic waiting. Because the *collect* processes communicate with a large number of p processes, we introduce a shorthand to denote a series of guards. We use $\overline{ctp.i.(j : 0 \leq j < N) \rightarrow S_j}$ as a shorthand for $\overline{ctp.i.0} \rightarrow S.0 \; [] \; \overline{ctp.i.1} \rightarrow S.1 \ldots [] \; \overline{ctp.i.(N-1)} \rightarrow S.(N-1)$.

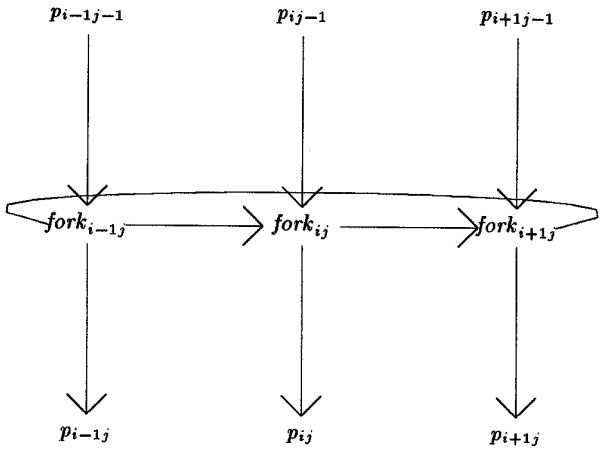

Figure 3.2: Interconnection pattern of processes

```
collect_i :: [ var m, l: node; w: bool;
              m := "arbitrary leaf"; w := true;
              do true → if ¬(ptc.i.(j : 0 ≤ j < N)) → ptc.i.j?l
                                                      if c.l < c.m → m := l; w := true
                                                      [] c.l ≥ c.m → skip
                                                      fi
                           [] ¬(ctp.i.(j : 0 ≤ j < N)) → ctp.i.j!c.m
                           [] ¬btc.i                   → btc.i?l;
                                                      if c.l < c.m → m := l; w := true
                                                      [] c.l ≥ c.m → skip
                                                      fi
                           [] w ∧ ¬ctb.i               → ctb.i!m; w := false
                           fi
              od
            ]
```

So far there is no communication between the p_{ij}'s for different values of i, that is, there is no exchange of work between the processors. The reason for introducing the communications was to have more control over the parameter passing; we exploit that now. Between every p_{ij-1} and p_{ij} we place a process $fork_{ij}$ These *fork* processes are connected in rings just like the *collect* processes. $fork_{ij}$ accepts nodes from p_{ij-1} and transmits these nodes to either p_{ij} or $fork_{i+1j}$, whichever is first in time (see figure 3.2). In this way work can be transported from one processor to neighboring processors.

In general, interprocessor communication is more expensive than intraprocessor communication. We want to design the *fork* processes in such a way that the amount of interprocessor communication is small. We require that a *fork* process only ask its neighbor in the ring for work if it will never receive anything from its predecessor in the same processor. As a result, when a node is transferred from one processor to another, the subtree rooted in this node is

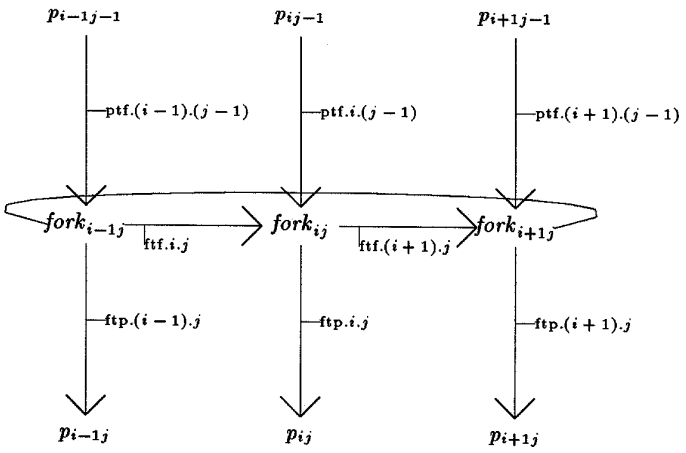

Figure 3.3: Interconnection pattern and channel names

maximal with respect to all subtrees that are still to be processed. This will minimize interprocessor communication since it will simply last longer before a processor has to ask its neighbor again.

We can associate the processes p_{ij} for fixed j with a level in the tree. From the above requirement, it is necessary to detect that the computation of a whole level has terminated. If this is detected, it is reported to the *fork* processes of the next level. These *fork* processes then can ask their neighbors for work.

For termination detection two new elements (fake nodes) are added to X: TERMl and TERMt. TERMt is used as a token to detect the termination of an entire level. TERMl is used to signal to the next level that the current level has terminated. TERMt is passed in the ring $fork_{ij}$ for fixed j. The token is generated by $fork_{0j}$ on receipt of TERMl from the previous level. If all *fork* processes have received the token twice then the computation of this level has terminated. Each process sends TERMl to the next level and termination detection on that level commences.

$fork_{ij}$ maintains a local variable *term* which is equal to the number of times TERMt has been received. The process terminates if *term* becomes 2. The (local) boolean r indicates whether information must be read from the ring. Since $fork_{0j}$ generates the token and consumes it after termination detection, the program for $fork_{0j}$ is slightly different. The *fork* processes communicate using the square array of channels *ftf*. The channels *ptp* are replaced by two square arrays *ptf* ("p to *fork*") and *ftp* ("*fork* to p"). (see figure 3.3). A local variable n is used to copy the input to the selected output.

```
fork_ij :: [ var n: node; term: integer; r: bool;
        term := 0; r := false;
        do term ≠ 2 → if r   → ftf.i.j?n
                      [] ¬r → ptf.i.(j − 1)?n
                      fi;
                      if n ≠ TERMt ∧ n ≠ TERMl → if ftf.(i + 1).j → ftf.(i + 1).j!n
                                                 [] ftp.i.j       → ftp.i.j!n
                                                 fi
                      [] n = TERMl → r := true;
                                     if i = 0 → ftf.(i + 1).j!TERMt
                                     [] i ≠ 0 → skip
                                     fi
                      [] n = TERMt → term := term + 1;
                                     if i = 0 ∧ term = 2 → skip
                                     [] i ≠ 0 ∨ term ≠ 2 → ftf.(i + 1).j!TERMt
                                     fi
                      fi
        od;
        ftp.i.j!TERMl
      ]
```

Invariant is for fixed i and j:

$term_{ij}$ = number of times $fork_{ij}$ has received TERMt
$0 \leq term_{ij} \leq 2$.

Since the termination token that is passed in the ring is generated by $fork_{0j}$ we conclude

$$\forall(i : 0 \leq i < M : term_{0j} \leq term_{ij})$$

hence

$$term_{0j} = 2 \;\Rightarrow\; \forall(i : 0 < i < M : term_{ij} = 2).$$

This means that if $fork_{0j}$ transmits TERMl to p_{0j+1} all $fork_{ij}$ have terminated and have sent or are sending TERMl to the next level. The process that has TERMt commits to the communication of TERMt (it does not wait for its neighbor to request it). This means that, when the token has been generated, always one process is ready to communicate it. If all $fork_{ij}$ have received TERMl from the previous level then eventually $term_{0j}$ will become 2, because $term_{0j} = 1$ implies that there is a process $fork_{ij}$ that tries to communicate TERMt.

Finally, we modify the processes p_{ij} such that the termination detection is taken into account.

```
p_ij :: [ var n: node; tm: real;
      tm := ∞; ftp.i.j?n;
      do n ≠ TERMl → if s.n = ∅ → if c.n < tm → ptc.i.j!n; tm := c.n
                                  [] c.n ≥ tm → skip
                                  fi
                   [] s.n ≠ ∅ → if c.n < tm → ctp.i.j?tm
                                  [] c.n ≥ tm → skip
                                  fi
                              ; if c.n < tm → forseq k ∈ s.n do ptf.i.j!k od
                                  [] c.n ≥ tm → skip
                                  fi
                   fi;
                   ftp.i.j?n
      od;
      if j < N − 1 → ptf.i.j!TERMl
      [] j = N − 1 → skip
      fi
    ]
```

The program is initialized by communicating via one of the channels $ptf.i.(-1)$ the root of the tree and communicating TERMl via all other channels $ptf.i.(-1)$.

This concludes our derivation of the parallel algorithm. The algorithm is the same for all processors; there is no "supervisor" (except for the asymmetry that is needed to start the termination detection). It dynamically balances the load on the network by communication between neighboring processors only. In the next section we give an example of a problem that can be solved using this algorithm.

3.4 The 0/1 knapsack problem

The 0/1 knapsack problem is stated as follows. Given are N pairs of numbers, $(p.i, w.i), 0 \leq i < N$ and a number $maxw$. The problem is to determine a set $S \subseteq \{0, 1, \ldots, N \quad 1\}$ such that $\Sigma(i : i \in S : p.i)$ is maximal under the restriction that $\Sigma(i : i \in S : w.i) \leq maxw$. The $p.i$ are called profits and the $w.i$ weights.

In order to use the above algorithm we must specify the parameters of the problem (X, s, c and r in definition 9). A solution can be described by a vector $x \in \{0,1\}^N$ with $x.i = 1 \equiv i \in S$. We use the notation $x(i : expr)$ to denote the vector x with $x.i$ replaced by $expr$.

$$X = \{0,1\}^N \times \{0, 1, \ldots, N\}$$
$$r = ((0, 0, \ldots, 0), 0)$$
$$s.(x,i) = \begin{cases} \{(x(i:0), i+1), (x(i:1), i+1)\} & \text{if } i < N \\ \emptyset & \text{if } i = N \end{cases}$$

The definition of c is more difficult. Notice that this is a maximization problem, and so c must be non-increasing. Assume that the pairs $(p.i, w.i)$ are ordered in decreasing order of $\frac{p.i}{w.i}$. If the

current solution (x,i) has weight w, then an upper bound for the maximum profit reached is $\Sigma(j:0\leq j<i:(x.j)p.j)+(maxw-w)\frac{p.i}{w.i}$) (since $\frac{p.i}{w.i}$ gives the highest profit per weight unit). With the additional definition $p.N=0$, we define c as follows.

$$c.(x,i)=\begin{cases}\Sigma(j:0\leq j<i:(x.j)p.j)+(maxw-\Sigma(j:0\leq j<i:(x.j)w.j))\frac{p.i}{w.i} \\ \qquad \text{if } \Sigma(j:0\leq j<i:(x.j)w.j)\leq maxw \\ -\infty \quad \text{otherwise}\end{cases}$$

The problem has the following property: in any node we can evaluate the cost of one specific leaf of that node. A leaf of (x,i) is (x,N) with cost $\Sigma(j:0\leq j<i:(x.j)p.j)$ (because we start with the zero vector, the elements $x.j, i\leq j<N$, must be 0). This is of importance for the parallel algorithm because the pruning is more efficient if the temporary maximum is high as soon as possible.

We have applied the above solution to this problem and implemented it in Occam on Inmos Transputers. In this implementation, we made the ring of *collect* processes bigger: we had a *collect* process for every p_{ij}. We did this because the big selection statement in the *collect* process turned out to perform rather badly. We found that the parallel implementation on 5 or 6 Transputers has roughly the same speed as the sequential algorithm on 1 Transputer. We used up to 17 Transputers.

The results of running the algorithm were rather peculiar. When the number of processors was doubled, the speedup more than doubled. In view of property 3 this implies that $B.1.n > B.p.n$ or, in other words, the computation time decreases with p.

This can be understood if one realizes that as more processors are added, the algorithm becomes more and more a mixture of depth-first and breadth-first search. Hence the number of nodes considered varies with the number of processors used. A similar phenomenon has been described in [35].

The main difference with the algorithm presented in [21], is that our algorithm dynamically balances the load on the network with little communication overhead between the processors. Furthermore, in our algorithm a new maximum is available to all processors as soon as possible.

Chapter 4

Some lattice theory

In this chapter, we recall some definitions and results from lattice theory. It is, in no respect, complete; we restrict ourselves to the parts that we need. Most of the material is a restatement of parts of [9] in a slightly more general context. We add some notions to handle cartesian product spaces. For a general discussion of lattice theory we refer to [4, 13, 14].

Definition 11 *A partial order on a set X, $\leq$, is a relation on X that is reflexive, anti-symmetric and transitive, i.e., for $x, y \in X$,*

$$\begin{aligned} &(i) \quad x \leq x, \\ &(ii) \quad x \leq y \wedge y \leq x \;\Rightarrow\; x = y, \\ &(iii) \quad x \leq y \wedge y \leq z \;\Rightarrow\; x \leq z. \end{aligned}$$

$(X, \leq)$ is called a poset.

□

Definition 12 *Let $(X, \leq)$ be a poset and $C \subseteq X$. $l \in X$ is called the greatest lower bound (glb) of C if*

$$\begin{aligned} &(i) \quad \forall(c : c \in C : l \leq c), \\ &(ii) \quad \forall(z : z \in X \wedge \forall(c : c \in C : z \leq c) : z \leq l). \end{aligned}$$

$u \in X$ is called the least upper bound (lub) of C if

$$\begin{aligned} &(i) \quad \forall(c : c \in C : c \leq u), \\ &(ii) \quad \forall(z : z \in X \wedge \forall(c : c \in C : c \leq z) : u \leq z). \end{aligned}$$

The glb of C is denoted by $\sqcap C$; the lub of C by $\sqcup C$.

□

Property 13 *If they exist, the greatest lower bound and the least upper bound of a subset of a poset $(X, \leq)$ are unique.*

Proof. (Only for the glb.) Let g and h be greatest lower bounds of $C \subseteq X$. From definition 12(ii) we have $g \leq h \wedge h \leq g$. From the anti-symmetry it follows that $g = h$.

□

Definition 14 *A sequence $x_i, 0 \leq i$ is called an ascending chain if* $\forall(i : 0 \leq i : x_i \leq x_{i+1})$ *and a descending chain if* $\forall(i : 0 \leq i : x_{i+1} \leq x_i)$.

□

In general, we are interested only in ascending chains. This is reflected in the following definition.

Definition 15 $(X, \leq)$ *is called a complete partial ordered set (cpo) if*

(i) *there exists a least element, $\bot$, in X, i.e.,* $\forall(x : x \in X : \bot \leq x)$,

(ii) *every ascending chain has a least upper bound.*

□

In this monograph, we will not use cpo's to a great extent. Instead we use the more restrictive concept of a complete lattice.

Definition 16 $(X, \leq)$ *is a complete lattice if every subset of X has a lub and a glb.*

□

From now on, $(X, \leq)$ is a complete lattice. From definition 16 it follows that X has a glb and a lub. We call them $\bot$ and $\top$ respectively. Notice, because of definition 2(ii), $\sqcap\emptyset = \top, \sqcup\emptyset = \bot$. For a set of two elements, we do not write $\sqcap\{x, y\}$ but $x \sqcap y$ because of the next property.

Property 17 *Both $\sqcap$ and $\sqcup$ define a function $X \times X \rightarrow X$ that is idempotent, commutative and associative, i.e., we have for $x, y, z \in X$,*

(i) $x \sqcap x = x$,

(ii) $x \sqcap y = y \sqcap x$,

(iii) $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$.

Similarly,

(iv) $x \sqcup x = x$,

(v) $x \sqcup y = y \sqcup x$,

(vi) $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.

Proof. We prove (iii). (vi) is similar and the other properties follow immediately from the definition.

$$
\begin{array}{ll}
 & a = (x \sqcap y) \sqcap z \\
\Rightarrow & \quad \{\text{definition } 12(i)\} \\
 & a \leq x \sqcap y \wedge a \leq z \\
\Rightarrow & \quad \{\text{definition } 12(i)\} \\
 & a \leq x \wedge a \leq y \wedge a \leq z \\
\Rightarrow & \quad \{\text{definition } 12(ii)\} \\
 & a \leq \sqcap\{x, y, z\}
\end{array}
$$

Furthermore,

$$
\begin{array}{ll}
 & b = \sqcap\{x, y, z\} \\
\Rightarrow & \quad \{\text{definition } 12(i), (ii)\} \\
 & b \leq x \wedge b \leq y \sqcap z \\
\Rightarrow & \quad \{\text{definition } 12(ii)\} \\
 & b \leq x \sqcap (y \sqcap z).
\end{array}
$$

Hence, $(x \sqcap y) \sqcap z \ \leq \ x \sqcap (y \sqcap z)$. The other inequality follows in the same way.

□

Property 18 *For $x, y \in X$ we have*

$$
\begin{array}{ll}
(i) & x \sqcap y = x \ \equiv \ x \leq y \\
(ii) & x \sqcup y = y \ \equiv \ x \leq y
\end{array}
$$

Proof. (Only (i).)

$$
\begin{array}{ll}
 & x \sqcap y = x \\
\Rightarrow & \quad \{\text{definition glb}\} \\
 & x \leq x \wedge x \leq y \\
\Rightarrow & \quad \{\text{definition glb}\} \\
 & x \leq x \sqcap y \\
\Rightarrow & \quad \{x \sqcap y \leq x, \text{ anti-symmetry}\} \\
 & x \sqcap y = x
\end{array}
$$

□

Corollary 19 *By first choosing $\top$ for y and then $\bot$ for x in property 18 we have for all $x, y \in X$,*

$$
\begin{array}{ll}
(i) & x \sqcap \top = x \ \wedge \ x \sqcup \top = \top, \\
(ii) & \bot \sqcap y = \bot \ \wedge \ \bot \sqcup y = y.
\end{array}
$$

□

In order to make the formulae more readable we leave the range $x, y, z \in X$ in quantified expressions with x, y or z as bound variables implicit. If we discuss functions from one lattice to another, we use only one symbol to denote the partial order. It will be clear from the context which partial order is meant.

Definition 20 *Let L_0 and L_1 be complete lattices, $W \subseteq L_0$, $V \subseteq W$, $x, y \in W$ and $f : L_0 \to L_1$. f is called*

$$
\begin{array}{ll}
\textit{universally conjunctive over } W & \textit{if } f.\sqcap V = \sqcap f.V, \\
\textit{positively conjunctive over } W & \textit{if } 1 \leq |V| \Rightarrow f.\sqcap V = \sqcap f.V, \\
\textit{finitely conjunctive over } W & \textit{if } 1 \leq |V| < \infty \Rightarrow f.\sqcap V = \sqcap f.V,
\end{array}
$$

universally disjunctive over W	*if* $f.\sqcup V = \sqcup f.V$,
positively disjunctive over W	*if* $1 \leq \|V\| \Rightarrow f.\sqcup V = \sqcup f.V$,
finitely disjunctive over W	*if* $1 \leq \|V\| < \infty \Rightarrow f.\sqcup V = \sqcup f.V$,
monotonic over W	*if* $x \leq y \Rightarrow f.x \leq f.y$.

The above definition is to be understood for arbitrary V, x and y. If it is clear from the context which set W is meant, we omit the restriction "over W" but simply say: f is universally conjunctive, finitely conjunctive, monotonic, etc. We refer to all these properties together as junctivity properties.

□

Property 21 *For a set W, universal conjunctivity implies positive conjunctivity which, in turn, implies finite conjunctivity. Similarly, for disjunctivity. Both finite conjunctivity and finite disjunctivity imply monotonicity.*

Proof. Except for monotonicity, the relations between the junctivity types follow immediately from the definition. We show that finite disjunctivity implies monotonicity. Choose $x, y \in W, x \leq y$.

$$
\begin{array}{ll}
 & f.x \leq f.y \\
= & \quad \{\text{property } 18(ii) : x \leq y \equiv x \sqcup y = y\} \\
 & f.x \leq f.(x \sqcup y) \\
= & \quad \{f \text{ finitely disjunctive over } W\} \\
 & f.x \leq f.x \sqcup f.y \\
= & \quad \{\text{definition}\} \\
 & true
\end{array}
$$

□

Theorem 22 *For f a monotonic function on X,*

(i) $f.\sqcap\{x : f.x \leq x : x\} \leq \sqcap\{x : f.x \leq x : x\}$,

(ii) $\sqcup\{x : x \leq f.x : x\} \leq f.\sqcup\{x : x \leq f.x : x\}$.

Proof. (Only (i).)

$$
\begin{array}{ll}
 & true \\
= & \quad \{\text{definition glb}\} \\
 & \forall(y : f.y \leq y : \sqcap\{x : f.x \leq x : x\} \leq y) \\
\Rightarrow & \quad \{f \text{ monotonic}\} \\
 & \forall(y : f.y \leq y : f.\sqcap\{x : f.x \leq x : x\} \leq f.y) \\
\Rightarrow & \quad \{\text{use domain expression, transitivity}\} \\
 & \forall(y : f.y \leq y : f.\sqcap\{x : f.x \leq x : x\} \leq y) \\
\Rightarrow & \quad \{\text{definition glb}\} \\
 & f.\sqcap\{x : f.x \leq x : x\} \leq \sqcap\{x : f.x \leq x : x\}
\end{array}
$$

□

Theorem 23 *(Knaster-Tarski) For f a monotonic function on X we have the following equalities.*

$$\begin{array}{ll} (i) & \sqcap\{x : f.x = x : x\} = \sqcap\{x : f.x \leq x : x\} \\ & \sqcup\{x : f.x = x : x\} = \sqcup\{x : x \leq f.x : x\} \\ (ii) & f.\sqcap\{x : f.x = x : x\} = \sqcap\{x : f.x = x : x\} \\ & f.\sqcup\{x : f.x = x : x\} = \sqcup\{x : f.x = x : x\} \end{array}$$

Notice that, because X is a complete lattice, all these upper and lower bounds exist. We call $\sqcap\{x : f.x = x : x\}$ the least fixpoint of f and $\sqcup\{x : f.x = x : x\}$ the greatest fixpoint of f.

Proof. We only deal with the lower bounds. Because $f.x = x \Rightarrow f.x \leq x$, we have

$$\sqcap\{x : f.x \leq x : x\} \leq \sqcap\{x : f.x = x : x\}.$$

If we show that $\sqcap\{x : f.x \leq x : x\}$ is a fixpoint of f, we also have the other inequality. (ii) then follows immediately.

$$\begin{array}{ll} & f.\sqcap\{x : f.x \leq x : x\} = \sqcap\{x : f.x \leq x : x\} \\ = & \quad \{\text{theorem } 22(i)\} \\ & \sqcap\{x : f.x \leq x : x\} \leq f.\sqcap\{x : f.x \leq x : x\} \\ \Leftarrow & \quad \{f.\sqcap\{x : f.x \leq x : x\} \text{ as instance for } x \text{ in } f.x \leq x, \text{ definition glb}\} \\ & f.(f.\sqcap\{x : f.x \leq x : x\}) \leq f.\sqcap\{x : f.x \leq x : x\} \\ \Leftarrow & \quad \{f \text{ monotonic}\} \\ & f.\sqcap\{x : f.x \leq x : x\} \leq \sqcap\{x : f.x \leq x : x\} \\ = & \quad \{\text{theorem } 22(i)\} \\ & \mathit{true} \end{array}$$

□

Remark. We can look at theorem 23 in a bit different way. The expression $f.x = x$ may be viewed as an equation in unknown x. The solution set of this equation has a glb and a lub because X is a complete lattice. The same holds for the solution sets of $x \leq f.x$ and $f.x \leq x$. Theorem 23 states that the glb of the solution set of $f.x = x$ is the same as the glb of the solution set of $f.x \leq x$ and that the lub of the solution set of $f.x = x$ is the same as the lub of the solution set of $x \leq f.x$. Moreover, both the glb and the lub of the solution sets of $f.x = x$ are solutions of $f.x = x$. We call the glb the smallest and the lub the greatest solution. This is the way we shall use theorem 23 in this monograph.

Definition 24 *For $x \in X^n$ and $0 \leq i < n$, $x.i$ is the ith component of x. We extend $\leq$ to X^n in the following way. For $x, y \in X^n$,*

$$x \leq_n y = \forall(i : 0 \leq i < n : x.i \leq y.i).$$

□

Theorem 25 *$(X^n, \leq_n)$ is a complete lattice. For $C \subseteq X^n$ we have $\sqcap_n C$ and $\sqcup_n C \in X^n$ as follows.*

$$(\sqcap_n C).i = \sqcap\{c : c \in C : c.i\}$$
$$(\sqcup_n C).i = \sqcup\{c : c \in C : c.i\}$$

Proof. We have to show that $\leq_n$ is a partial order on X^n and that every subset of X^n has this lub and glb. This follows directly from the definition.

□

We denote $\sqcap_n X^n$ by $\bot^n$, i.e., a vector of n $\bot$'s. Similarly, $\sqcup_n X^n = \top^n$.

Theorem 26 *For $I \subseteq \{0, 1, \ldots, n-1\}$ and $x \in X^n$,*

$$(\{y : y \in X^n \wedge \forall(i : 0 \leq i < n : i \in I \vee x.i = y.i) : y\}, \leq_n)$$

is a complete lattice.

Proof. This also follows directly from the definitions. Notice that we have in fact, for every x, a copy of $X^{|I|}$.

□

Definition 27 *Let f be a function from X^n to X and $I \subseteq \{0, 1, \ldots, n-1\}$. $W \subseteq X^n$ is called an I-projection if every two elements in W differ only in components with an index in I. f is called I-junctive if f is junctive over every I-projection.*

□

Notice that every subset of an I-projection is also an I-projection. In the following, let $I \subseteq \{0, 1, \ldots, n-1\}$.

Theorem 28 *Let $f : X^n \to X$.*

$$f \text{ is } I\text{-monotonic} \quad \equiv \quad \forall(i : i \in I : f \text{ is } \{i\}\text{-monotonic})$$

("Monotonicity is componentwise.")

Proof. ($\Rightarrow$) Clearly, for all $i \in I$, every $\{i\}$-projection is an I-projection. Hence, f is monotonic over every $\{i\}$-projection.
($\Leftarrow$) Choose an I-projection W and $x, y \in W$ such that $x \leq_n y$. Define a series, $a_k \in X^n, 0 \leq k \leq n$ as follows.

$$a_k.i = \begin{cases} x.i & \text{if } i \geq k \\ y.i & \text{if } i < k \end{cases}$$

Then, $a_0 = x, a_n = y$ and, for $0 \leq k < n$, a_k and a_{k+1} differ in at most one component, viz. k, and if they differ then $k \in I$. Since $x \leq_n y$, $a_k \leq_n a_{k+1}$. From the above and from the reflexivity, it follows that f is monotonic over $\{a_k, a_{k+1}\}$. Hence, $f.a_k \leq_n f.a_{k+1}$ for $0 \leq k < n$ and by transitivity, $f.x \leq_n f.y$.

□

Theorem 29 *Let $f : X^n \to X$ be I-monotonic. Then, for every I-projection V,*

$$f.\sqcap_n V \leq \sqcap f.V.$$

Proof. Notice that, since X^n is a complete lattice, $\sqcap_n V$ exists. Moreover, an arbitrary element of V and $\sqcap_n V$ can differ only at indices in I. Hence, $V \cup \{\sqcap_n V\}$ is an I- projection.

$$\begin{array}{ll} & f.\sqcap_n V \leq \sqcap f.V \\ \Leftarrow & \quad \{\text{definition glb}\} \\ & \forall(v : v \in f.V : f.\sqcap_n V \leq v) \\ = & \quad \{f.V = \{v : v \in V : f.v\}, \text{renaming the dummy}\} \\ & \forall(v : v \in V : f.\sqcap_n V \leq f.v) \\ \Leftarrow & \quad \{\text{domain split}\} \\ & \forall(v : v \in V \cup \{\sqcap_n V\} : f.\sqcap_n V \leq f.v) \\ \Leftarrow & \quad \{f \text{ monotonic over } V \cup \{\sqcap_n V\}\} \\ & \forall(v : v \in V \cup \{\sqcap_n V\} : \sqcap_n V \leq v) \\ = & \quad \{\text{definition glb, reflexivity}\} \\ & \mathit{truc} \end{array}$$

□

In the following, let f be a function from X^{n+1} to X and define $g, h : X^n \to X$ as follows.

$$g.x = \sqcap\{y : f.x.y = y : y\}$$
$$h.x = \sqcup\{y : f.x.y = y : y\}$$

Furthermore, let $I \subseteq \{0, 1, \ldots, n-1\}$.

Theorem 30 *If f is $(I \cup \{n\})$-monotonic, g and h are I-monotonic.*

Proof. Theorem 28 gives that f is $\{n\}$-monotonic, hence, by theorem 26, we may apply theorem 23. We can therefore characterize g by

$$f.x.(g.x) = g.x, \tag{4.1}$$
$$f.x.y \leq y \Rightarrow g.x \leq y, \text{ for all } y \in X, \tag{4.2}$$

and h by

$$f.x.(h.x) = h.x, \tag{4.3}$$
$$y \leq f.x.y \Rightarrow y \leq h.x, \text{ for all } y \in X. \tag{4.4}$$

Choose x and x' in X^n such that $\{x, x'\}$ is an I-projection, and $x \leq_n x'$.

$$\begin{array}{ll} & g.x \leq g.x' \\ \Leftarrow & \quad \{(4.2)\} \\ & f.x.(g.x') \leq g.x' \end{array}$$

$$
\begin{array}{ll}
= & \{(4.1)\} \\
& f.x.(g.x') \leq f.x'.(g.x') \\
\Leftarrow & \{\{x, x'\} \times \{g.x'\} \text{ is an } I\text{-projection, } f \text{ is } I \cup \{n\}\text{-monotonic}\} \\
& x \leq_n x'.
\end{array}
$$

Furthermore, for h,

$$
\begin{array}{ll}
& h.x \leq h.x' \\
\Leftarrow & \{(4.4)\} \\
& h.x \leq f.x'.(h.x) \\
= & \{(4.3)\} \\
& f.x.(h.x) \leq f.x'.(h.x) \\
\Leftarrow & \{\{x, x'\} \times \{h.x\} \text{ is an } I\text{-projection, } f \text{ is } I \cup \{n\}\text{-monotonic}\} \\
& x \leq_n x'.
\end{array}
$$

□

Theorem 31 *If f is $(I \cup \{n\})$-universally conjunctive, h is I-universally conjunctive.*

Proof. Notice that from f's conjunctivity property it follows that f is $(I \cup \{n\})$-monotonic. Hence, by theorem 30, h is I-monotonic. Choose an I-projection, W, and $V \subseteq W$. V is also an I-projection.

$$
\begin{array}{ll}
& h.\sqcap_n V = \sqcap h.V \\
= & \{h \text{ is monotonic over } V\text{: theorem 29, antisymmetry}\} \\
& \sqcap h.V \leq h.\sqcap_n V \\
\Leftarrow & \{(4.4)\} \\
& \sqcap h.V \leq f.(\sqcap_n V).(\sqcap h.V) \\
= & \{\text{definition 24, theorem 25}\} \\
& \sqcap h.V \leq f.\sqcap_{n+1} \{v : v \in V : (v, h.v)\} \\
= & \{\{v : v \in V : (v, h.v)\} \text{ is } (I \cup \{n\})\text{-projection, } f \ (I \cup \{n\})\text{-universally conjunctive}\} \\
& \sqcap h.V \leq \sqcap\{v : v \in V : f.v.(h.v)\} \\
= & \{(4.3), h.V = \{v : v \in V : h.v\}\} \\
& \sqcap h.V \leq \sqcap h.V \\
= & \{\text{reflexivity}\} \\
& \mathit{true}
\end{array}
$$

□

For our final result, we need to restrict the set X and the partial order $\leq$.

Definition 32 *A complete lattice $(L, \leq)$ is called distributive if for each $x \in L$ and $C \subseteq L$,*

(i) $x \sqcap (\sqcup C) = \sqcup\{c : c \in C : x \sqcap c\}$,

(ii) $x \sqcup (\sqcap C) = \sqcap\{c : c \in C : x \sqcup c\}$.

□

Definition 33 *A complete lattice $(L, \leq)$ is called complemented if, for every $x \in L$, there exists a $y \in L$, called a complement of x, such that*

(i) $x \sqcap y = \bot$,
(ii) $x \sqcup y = \top$.

□

The notions of distributivity and complement for a complete lattice $(L, \leq)$, are naturally generalized to $(L^n, \leq_n)$.

Theorem 34 *In a complete, distributive, complemented lattice $(L, \leq)$, the complement of $x \in L$ is uniquely determined.*

Proof. Choose $x \in L$ and let y and z be complements of x.

$$
\begin{array}{ll}
 & y \\
= & \quad \{\text{property 18(ii)}\} \\
 & y \sqcup \bot \\
= & \quad \{z \text{ is a complement}\} \\
 & y \sqcup (x \sqcap z) \\
= & \quad \{\text{distributivity}\} \\
 & (y \sqcup x) \sqcap (y \sqcup z) \\
= & \quad \{y \text{ is a complement}\} \\
 & \top \sqcap (y \sqcup z) \\
= & \quad \{z \text{ is a complement}\} \\
 & (x \sqcup z) \sqcap (y \sqcup z) \\
= & \quad \{\text{distributivity}\} \\
 & (x \sqcap y) \sqcup z \\
= & \quad \{y \text{ is a complement}\} \\
 & \bot \sqcup z \\
= & \quad \{\text{property } 18(ii)\} \\
 & z
\end{array}
$$

□

A complete, distributive, complemented lattice is called a *boolean* lattice. From now on, $(X, \leq)$ is a boolean lattice and, hence, $(X^n, \leq_n)$ is that too. We denote the complement of $x \in X$ by $\sim x$.

Theorem 35 *For $x, y, z \in X$,*

$$x \sqcap y \leq z \;\equiv\; x \leq z \sqcup \sim y.$$

Proof.

$$
\begin{array}{ll}
 & x \sqcap y \leq z \\
= & \quad \{\text{property 18}\} \\
 & (x \sqcap y) \sqcup z = z \\
\Rightarrow & \quad \{\text{Leibniz}\} \\
 & (x \sqcap y) \sqcup z \sqcup {\sim}y = z \sqcup {\sim}y \\
= & \quad \{\text{distributivity}\} \\
 & (x \sqcup z \sqcup {\sim}y) \sqcap (z \sqcup y \sqcup {\sim}y) = z \sqcup {\sim}y \\
= & \quad \{y \sqcup {\sim}y = \top, \text{ corollary 19}\} \\
 & x \sqcup z \sqcup {\sim}y = z \sqcup {\sim}y \\
= & \quad \{\text{property 18}\} \\
 & x \leq z \sqcup {\sim}y \\
= & \quad \{\text{property 18}\} \\
 & x \sqcup z \sqcup {\sim}y = z \sqcup {\sim}y \\
\Rightarrow & \quad \{\text{Leibniz}\} \\
 & (x \sqcup z \sqcup {\sim}y) \sqcap y = (z \sqcup {\sim}y) \sqcap y \\
= & \quad \{\text{distributivity}\} \\
 & ((x \sqcup z) \sqcap y) \sqcup ({\sim}y \sqcap y) = (z \sqcap y) \sqcup ({\sim}y \sqcap y) \\
= & \quad \{{\sim}y \sqcap y = \bot, \text{ corollary 19}\} \\
 & (x \sqcup z) \sqcap y = z \sqcap y \\
= & \quad \{\text{distributivity}\} \\
 & (x \sqcap y) \sqcup (z \sqcap y) = z \sqcap y \\
= & \quad \{\text{property 18}\} \\
 & x \sqcap y \leq z \sqcap y \\
\Rightarrow & \quad \{\text{definition glb}\} \\
 & x \sqcap y \leq z
\end{array}
$$

□

Now we are ready for our final theorem.

Theorem 36 *If f is $(I \cup \{n\})$-finitely conjunctive, then, for every I-projection $\{x, y\}$,*

$$g.(x \sqcap_n y) = g.x \sqcap h.y.$$

Proof. We prove two inequalities. f is $\{n\}$-monotonic, hence, by theorem 26, theorem 23 applies. Let $\{x, y\}$ be an I-projection in X^n.

$$
\begin{array}{ll}
 & g.(x \sqcap_n y) \leq g.x \sqcap h.y \\
\Leftarrow & \quad \{(4.2)\}
\end{array}
$$

$$
\begin{array}{ll}
 & f.(x \sqcap_n y).(g.x \sqcap h.y) \leq g.x \sqcap h.y \\
= & \quad \{\{(x, g.x), (y, h.y)\} \text{ is an } (I \cup \{n\})\text{-projection, } f \text{ is } (I \cup \{n\})\text{-conjunctive}\} \\
 & f.x.(g.x) \sqcap f.y.(h.y) \leq g.x \sqcap h.y \\
= & \quad \{(4.1), (4.3)\} \\
 & g.x \sqcap h.y \leq g.x \sqcap h.y \\
= & \quad \{\text{reflexivity}\} \\
 & true
\end{array}
$$

As a result, we obtain also

$$g.(x \sqcap_n y) \leq h.y. \tag{4.5}$$

Next, the other inequality.

$$
\begin{array}{ll}
 & g.x \sqcap h.y \leq g.(x \sqcap_n y) \\
= & \quad \{\text{theorem 35}\} \\
 & g.x \leq g.(x \sqcap_n y) \sqcup \sim h.y \\
\Leftarrow & \quad \{(4.2)\} \\
 & f.x.(g.(x \sqcap_n y) \sqcup \sim h.y) < g.(x \sqcap_n y) \sqcup \sim h.y \\
= & \quad \{\text{theorem 35}\} \\
 & f.x.(g.(x \sqcap_n y) \sqcup \sim h.y) \sqcap h.y \leq g.(x \sqcap_n y) \\
= & \quad \{(4.3)\} \\
 & f.x.(g.(x \sqcap_n y) \sqcup \sim h.y) \sqcap f.y.(h.y) \leq g.(x \sqcap_n y) \\
= & \quad \{\{(x, g.(x \sqcap_n y) \sqcup \sim h.y), (y, h.y)\} \text{ is an } I \cup \{n\}\text{-projection}\} \\
 & f.(x \sqcap_n y).((g.(x \sqcap_n y) \sqcup \sim h.y) \sqcap h.y) \leq g.(x \sqcap_n y) \\
= & \quad \{\text{distributivity, } \sim h.y \sqcap h.y = \bot, \text{ corollary 19}\} \\
 & f.(x \sqcap_n y).(g.(x \sqcap_n y) \sqcap h.y) \leq g.(x \sqcap_n y) \\
= & \quad \{(4.5)\} \\
 & f.(x \sqcap_n y).(g.(x \sqcap_n y)) \leq g.(x \sqcap_n y) \\
= & \quad \{(4.1)\} \\
 & true
\end{array}
$$

□

Remark. We introduced distributivity and the complement mainly to give *this* proof, which is borrowed from the proof of (52), chapter eight, in [9]. We do not know whether theorem 36 holds for more general lattices. However, we conjecture that this is not the case. We do not need the monotonicity of h in the first part of the proof as in [9].

We apply this theory to the predicate calculus. The predicates over some space form a complete, distributive, complemented lattice with the following choices: implication over the state space for the partial order, universal quantification for the greatest lower bound and existential quantification for the least upper bound of a set and logical negation for the complement.

In the chapters six and seven of this monograph, we encounter functions of vectors of predicates. The junctivity properties of these functions are given only for a limited subset of the total argument. This is why we introduced the notion of a projection. Theorem 36 is used at the end of chapter six where some facts about the repetitive construct are proven.

Chapter 5

Defining properties of programs

5.1 Introduction

In this and the remaining chapters, we are primarily concerned with properties of programs. The programs under consideration are the ones that satisfy the syntax of Dijkstra's guarded command language ([8, 9]) hence we do not consider the parallel construct or the communication constructs. The reason for this restriction is that the parallel construct is not compositional. We come back to that in chapter eight.

A program can be executed by some kind of machine and we are interested in what happens if a program is fed into a machine. We therefore study properties of the machine behavior, associated with the execution of the program. Since there exist many different kinds of machines, we have to look for a suitable abstraction that still captures the essentials of the physical machines at hand. In a program we have two parts: variables and instructions to change these variables. The values of the variables are collected together in a so-called state and an execution of the (instructions of the) program consists of a sequence of modifications of this state. Every machine capable of executing such a program must have some way to represent the state and a means to change this state according to the program's instructions. An execution of the machine can be viewed as a sequence of states namely the sequence assumed by the machine during the interpretation of the program. We call such a sequence a *computation* of the program. The properties that we are interested in are properties of these computations.

Before we can study properties in this sense we have to give meaning to programs as syntactical objects. Since we want to study properties of the state sequences that may arise from executing a program, we must define for each program which state sequences may result from it.

Much work has been done in defining the meaning, or, what it is usually called, the semantics of this, or a related, programming language ([8, 9, 1]). Because in general the very reason of writing a program is to have it executed by a machine, it seems natural to choose the semantics described above as the meaning of the program. This semantics is called the *operational semantics.* This choice, however, has some severe disadvantages. In order to see this we need to be more explicit about our use of properties of programs.

A property of a program is a statement about the possible computations of the program that

is either true or false, hence, a property can be associated with a set of computations. In general, a program does not define one single computation. Which one is chosen highly depends on the value of the state in which the machine is started. Furthermore it is possible that the program contains a choice among computations. We then say that the program is *nondeterministic.* A program now has a certain property if all possible computations of the program are within that property.

We reason about programs and properties in different ways. Firstly, we use a property, or a number of properties, to specify a program. Secondly, given a program we verify that the program has a property. Thirdly, the properties specifying a program may be used to obtain a program text through calculation. With respect to these three ways of using properties, we make some remarks about the operational semantics.

In the specification of a program, we often restrict ourselves to a small class of properties. One of the most commonly used properties is the property that all computations terminate in a state satisfying some condition. The semantics of a program in terms of the state sequences may be overspecific with respect to the properties that we want to specify. For the above property, we are not interested at all in the states between the initial and the final state.

In verifying whether a program has a property, going through all possible computations is cumbersome and prone to errors. It is difficult to consider all possible computations, especially in the presence of nondeterminism. Finally, if a specification is used to derive a program, properties must be defined on the basis of the program text and it must be possible to judge that a certain program has a property, without referring to machine behavior.

It has been recognized that it is not necessary to reason about intermediate states of a computation if one's only interest is the final state. This has lead to different, more abstract, semantics. In [1], programs are viewed as mappings from initial states to final states. In [9] even the concept of a state has been dropped. A program is defined to be a function from predicates to predicates. If required, a state can enter the picture and a predicate is associated with the set of states in which it yields *true.* Associated with a program S are two functions, *wp* and *wlp* on predicates (pronounced weakest precondition and weakest liberal precondition respectively). $wp.S.p$ is the weakest precondition for S such that, if execution of S is started in a state satisfying this precondition, S terminates in a state satisfying p. $wlp.S.p$ is similar but also allows nontermination. In order to show that a program S has the property "S terminates in p", we show that the precondition of S, i.e., the condition that captures all states in which S may be started, implies $wp.S.p$. The operational interpretation of a program in terms of its computations is used in [9] only as an informal justification of the definitions. Instead of referring to the operational semantics to prove facts about these functions, some characteristic properties are isolated. The characteristics of *wp* and *wlp* are captured in three rules: $wlp.S$ distributes over universal quantification, $wp.S.false = false$ for all S and $wp.S.p = wp.S.true \wedge wlp.S.p$ for all S and p. For every new construct that is added to the language, *wlp* and *wp* must be defined and these characteristics have to be verified. All the theory developed using only these three characteristics automatically applies.

An advantage of defining the semantics of a program in this way is that only the relevant aspects, with respect to a certain property, are considered. In the above case, an implementer has freedom in implementing the language constructs. Furthermore, the operational semantics is but one model for this theory. There may be other models. A disadvantage is that attention is

focused on one particular property. For instance, if the relation between initial and final states is chosen to be *the* semantics of the program, other information is lost.

What kind of properties do we use to specify programs? For sequential programs, specifying the final state is often enough. However, there also exist useful programs that do not terminate, for instance, an operating system. For an operating system we are not at all interested in establishing a final condition; we want to specify what happens if we interact with it. (Programs like an operating system are sometimes called *reactive* programs.) Other examples are an editor and other interactive programs and, most importantly, parallel programs. If we consider the parallel programs of the previous chapters than we see that the processes in these programs may be stopped at many points because of communication and synchronization. In order to show the correctness of these programs, we have to prove that eventually such a communication will take place. The property "eventually some condition becomes *true*" is essentially different from the property "the program terminates in a state that satisfies some condition" in that in the former case, the condition may become *true* during execution of the program. It does not even require the program to terminate. Therefore, other semantics has been given, based on the operational semantics, for example, *temporal logic* ([25, 32]). Other properties and ways to define and prove these properties have been proposed. A property that describes the interaction with a system such as an operating system is *leads-to*. For predicates p and q, p *leads-to* q is a property of (all the computations of) a program if, whenever p holds during execution of the program, q holds also or will hold thereafter. This property has been defined in temporal logic for state sequences. In the program notation UNITY ([6]), *leads-to* has been defined on the basis of a program text. Other properties turn out to be important as well. For instance the property "*always* p", i.e., p holds at every step during the execution of a program, is an important property in parallel programming. We already mentioned "*ever* q" (or "eventually q"), expressing that q will hold at least once in every computation of the program.

We observe that we have many different properties that we want to define. This leaves us with the problem that we need some "basic semantics" that serves as a basis for all properties. In spite of the drawbacks mentioned above, we choose the operational semantics for this. Since we can conceive all kinds of properties, we cannot hope for a more abstract semantics than that. Besides this, we develop a mechanism that we can use more or less mechanically to define arbitrary properties of programs. This mechanism is based on a function called "weakest precondition", that is a generalization of the functions *wlp* and *wp* defined in [9].

In the next section we define this operational semantics. In the third section, we introduce the properties *ever*, *leads-to* and *always*, based on this semantics. The last section of this chapter is about non-determinism. In the next chapters we then investigate an axiomatic definition of these properties, i.e., a definition based on the program text not explicitly referring to the operational details.

5.2 Operational Semantics

When defining the operational semantics of a programming language we look very closely at what happens if a program, written in that language, is executed by a machine. The central idea is that the machine has an internal state that is changed according to the program's instructions.

This state consists of the values of the variables declared in the program. The semantics of a program is the set of all state sequences (finite or infinite) that may result from executing the program. Hence it is sufficient to restrict our attention to this state, the space in which it assumes its values and the sequences over this space.

We first introduce some notation.

V	=	The list of all program variables.
X	=	A cartesian product of domains, one for each program variable, in the order in which they occur in V. X is called the state space and its elements are called states. X is nonempty.
$Bool$	=	$\{true, false\}$.
P	=	The functions from X to $Bool$. Elements of P are called predicates.
T	=	The set of all nonempty finite or infinite sequences of states.
T'	=	The set of all finite or infinite sequences of states.

We use the names of program variables to select a component of a state. If y is a program variable such that $V.i = y$ (i.e., y is program variable i) then $x.y$, for $x \in X$, is by definition equal to $x.i$ (i.e., component i of x).

For $t \in T$, we use t^∞ to denote an infinite sequence of t's. For strings, we use juxtaposition to denote catenation. This is extended to sets in the obvious way. The catenation of a sequence to the tail of an infinite sequence is that infinite sequence. For a string s, $|s|$ denotes its length, $s.i$ element i if $0 \leq i < |s|$ and $last.s$ its last element if $|s| \prec \infty$. For strings s and t, $s \sqsubseteq t$ denotes that s is a prefix of t. $\sqsubseteq$ is a partial order on strings. Sometimes it is convenient to distinguish between finite and infinite strings. For A a set of strings, $fin.A$ is the subset of A consisting of the finite strings in A and $inf.A$ the remaining part of A.

P, equipped with the following order

$$p \leq q \;=\; \forall(x : x \in X : p.x \Rightarrow q.x),$$

is a boolean lattice, as introduced in the previous chapter. As mentioned in the introduction of this monograph, we use $[p]$ as a shorthand for $\forall(x : x \in X : p.x)$. Program variables occurring free in a predicate are used to select a component of the state on which the predicate is applied, as described above. We use the name *true* for the function that yields *true* everywhere and *false* for the function that yields *false* everywhere. In our examples, we use the traditional syntax to denote a predicate.

With these preliminaries, we can define programs. A program describes which sequences are possible computations, hence a program may be viewed as a subset of T. However, not all these subsets can be programs. A program is written in some language, in particular, Dijkstra's guarded command language. A grammar is used to define the basic language constructs and how these basic constructs may be combined into valid programs. We define recursively, for every syntactic construct in the language, which sequences may arise from starting execution of that construct. If a subset of T is to be viewed as a program, there must be, for every $x \in X$, a sequence in that subset that starts with x, since execution of a program can commence in an arbitrary initial state.

$$Prog = \{S : S \subseteq T \wedge \forall(x : x \in X : \exists(s : s \in S : s.0 = x)) : S\},$$
the set of programs.

Notice that, since X is nonempty, a program can never be the empty set.

We now look more closely at our programming language. We recall from the introduction the constructs and their intended meaning.

abort	- loop forever
skip	- do nothing
$y := e$	- assign expression e to program variable y
$S; U$	- sequential composition
if $[](i :: B_i \rightarrow S_i)$ **fi**	- execute an S_i for which B_i holds
do $B \rightarrow S$ **od**	- repeat S as long as B holds

All these constructs must be defined as elements of *Prog*. For every definition we have to verify whether we defined a proper element of *Prog*, i.e., we have to verify the restriction in the definition of *Prog*.

We start with the definitions of the three basic constructs.

$$abort = \{x : x \in X : x^\infty\}$$
$$skip = \{x : x \in X : xx\}$$

abort repeats the state in which it is started forever. *skip* does not do anything: it produces the same final state as its initial state.

In order to define the assignment statement, we introduce substitution. Let d be a function from X to the component of the state space in which some program variable y assumes its values.

$$x[y/d.x].z = \begin{cases} x.z & \text{if } y \neq z \\ d.x & \text{if } y = z \end{cases}$$

Using this, the assignment statement is defined by

$$y := e = \{x : x \in X : x(x[y/e.x])\}.$$

The assignment changes the state in which it is started in that at position y the value of expression $e.x$ is stored, that is, the value of e computed in state x. We do not bother here about type constraints, i.e, whether the type of e is appropriate to assign $e.x$ to y or whether e can be computed at all in the point x.
Clearly, all three constructs satisfy the constraint imposed in the definition of *Prog*.

We define sequential composition in a slightly more general context than only for members of *Prog*. We define it for general subsets of T. For $A, B \subseteq T$,

$$A; B = \{a, x, b : ax \in A \wedge b \in B \wedge b.0 = x : ab\} \cup inf.A.$$

Notice that, since T does not contain the empty sequence this is a proper definition. $A; B$ contains the infinite sequences in A and all sequences formed from a finite sequence $a \in A$ and a

sequence $b \in B$ such that the last element of a is the first element of b, by catenating a without its last element, and b. In the case that B is nonempty, we can omit the term *inf.A* because of our convention with respect to the catenation of infinite sequences.

Property 37 *Let* $A, B \subseteq T$.

$$(i) \quad \forall(s : s \in A; B : \exists(a : a \in A : a \sqsubseteq s))$$
$$(ii) \quad B \in \mathit{Prog} \Rightarrow \forall(a : a \in A : \exists(s : s \in A; B : a \sqsubseteq s))$$

Proof. (i) follows directly from the definition of ";". (ii) Choose $a \in A$. If $|a| = \infty$, $a \in A; B$. If $|a| < \infty$, there exists a $xb \in B, x \in X$ such that $a = cx$ since $B \in \mathit{Prog}$. From the definition of ";" it follows that $cxb \in A; B$. Obviously, $a \sqsubseteq cxb$.

□

From property 37, it follows that $A, B \in \mathit{Prog}$ implies $A; B \in \mathit{Prog}$.

Associated with a predicate $p \in P$, is a subset of X where p yields *true*. We can view this as a subset of T as well which we denote by p^{set}.

$$p^{set} = \{s : s \in T \wedge |s| = 1 \wedge p.(s.0) : s\}$$

Now we can define the alternative construct. We abbreviate **if** $[](i :: B_i \rightarrow S_i)$ **fi** by *IF*.

$$IF = \bigcup(i :: B_i^{set}; S_i) \cup (\forall(i :: \neg B_i))^{set}; \mathit{abort}$$

We have to show again that we defined a proper member of *Prog*. Notice that, for a predicate p, $p^{set}; S$ is the restriction of S to those sequences that start with a state for which p holds. Since for every $x \in X$ we have $\forall(i :: \neg B_i.x) \vee \exists(i :: B_i.x)$, the fact that *IF* is an element of *Prog* follows from the fact that S_i and *abort* are elements of *Prog*.

The last construct that we have to consider is the repetition. Executing the repetitive construct **do** $B \rightarrow S$ **od**, implies the repeated, sequential execution of the body, S, possibly an infinite number of times. We therefore first study some properties of the sequential composition.

Property 38 *Let* $A, B \subseteq T$.

$$(i) \quad \mathit{inf}.(A; B) = \mathit{inf}.A \cup (\mathit{fin}.A); \mathit{inf}.B$$
$$(ii) \quad \mathit{fin}.(A; B) = (\mathit{fin}.A); \mathit{fin}.B$$

Proof. Immediate from the definition.

□

Property 39 *For* $A, B, C \subseteq T$ *such that* A *and* B *contain no infinite sequences,*

$$(A; B); C = A; (B; C).$$

Proof. Notice that, according to property 38(i), $\mathit{inf}.(A;B)$ is also empty.

$$
\begin{array}{ll}
 & (A;B);C \\
= & \quad \{\text{definition ";"}, \mathit{inf}.(A;B) = \emptyset\} \\
 & \{u,x,c : ux \in A;B \wedge c \in C \wedge c.0 = x : uc\} \\
= & \quad \{\text{definition ";"}, \mathit{inf}.A = \emptyset\} \\
 & \{u,x,c : ux \in \{a,y,b : ay \in A \wedge b \in B \wedge b.0 = y : ab\} \wedge c \in C \wedge c.0 = x : uc\} \\
= & \quad \{\text{calculus, renaming the dummy}\} \\
 & \{u,x,c,a,y,b : ux = abx \wedge ay \in A \wedge bx \in B \wedge (bx).0 = y \wedge c \in C \wedge c.0 = x : uc\} \\
= & \quad \{\text{one point rule}\} \\
 & \{x,c,a,y,b : ay \in A \wedge bx \in B \wedge (bx).0 = y \wedge c \in C \wedge c.0 = x : abc\} \\
= & \quad \{\text{calculus}\} \\
 & \{a,y,v : ay \in A \wedge v \in \{b,x,c : bx \in B \wedge c \in C \wedge c.0 = x : bc\} \wedge v.0 = y : av\} \\
= & \quad \{\text{definition ";"}, \mathit{inf}.B = \emptyset\} \\
 & \{a,y,v : ay \in A \wedge v \in B;C \wedge v.0 = y : av\} \\
= & \quad \{\text{definition ";"}, \mathit{inf}.A = \emptyset\} \\
 & A;(B;C)
\end{array}
$$

□

Property 40 *";" distributes over arbitrary union in both arguments.*

Proof. Let I be a collection of subsets of T and $B \subseteq T$.

$$
\begin{array}{ll}
 & \bigcup(A : A \in I : A);B \\
= & \quad \{\text{definition ";"}\} \\
 & \{a,x,b : ax \in \bigcup(A : A \in I : A) \wedge b \in B \wedge b.0 = x : ab\} \cup \mathit{inf}.(\bigcup(A : A \in I : A)) \\
= & \quad \{\text{interchange unions, definition } \mathit{inf}\} \\
 & \bigcup(A : A \in I : \{a,x,b : ax \in A \wedge b \in B \wedge b.0 = x : ab\}) \cup \bigcup(A : A \in I : \mathit{inf}.A) \\
= & \quad \{\text{calculus}\} \\
 & \bigcup(A : A \in I : \{a,x,b : ax \in A \wedge b \in B \wedge b.0 = x : ab\} \cup \mathit{inf}.A) \\
= & \quad \{\text{definition}\} \\
 & \bigcup(A : A \in I : A;B)
\end{array}
$$

A similar proof can be given for the distributive property in the other argument.

□

Property 41 *";" is associative.*

Proof. Let $A, B, C \subseteq T$. We prove this property by showing $\mathit{inf}.((A;B);C) = \mathit{inf}.(A;(B;C))$ and $\mathit{fin}.((A;B);C) = \mathit{fin}.(A;(B;C))$.

$$\mathit{inf}.((A;B);C)$$

$$
\begin{aligned}
&= \quad \{\text{property } 38(i)\} \\
&\quad \mathit{inf}.(A;B) \cup \mathit{fin}.(A;B); \mathit{inf}.C \\
&= \quad \{\text{property } 38(i),(ii)\} \\
&\quad \mathit{inf}.A \cup (\mathit{fin}.A); \mathit{inf}.B \cup ((\mathit{fin}.A); \mathit{fin}.B); \mathit{fin}.C \\
&= \quad \{\text{property } 39\} \\
&\quad \mathit{inf}.A \cup (\mathit{fin}.A); \mathit{inf}.B \cup (\mathit{fin}.A); ((\mathit{fin}.B); \mathit{fin}.C) \\
&= \quad \{\text{property } 40\} \\
&\quad \mathit{inf}.A \cup (\mathit{fin}.A); (\mathit{inf}.B \cup (\mathit{fin}.B); \mathit{fin}.C) \\
&= \quad \{\text{property } 38(i)\} \\
&\quad \mathit{inf}.A \cup (\mathit{fin}.A); \mathit{inf}.(B;C) \\
&= \quad \{\text{property } 38(i)\} \\
&\quad \mathit{inf}.(A;(B;C))
\end{aligned}
$$

$$
\begin{aligned}
&\quad \mathit{fin}.((A;B);C) \\
&= \quad \{\text{property } 38(ii)\} \\
&\quad (\mathit{fin}.(A;B)); \mathit{fin}.C \\
&= \quad \{\text{property } 38(ii)\} \\
&\quad ((\mathit{fin}.A); \mathit{fin}.B); \mathit{fin}.C \\
&= \quad \{\text{property } 39\} \\
&\quad (\mathit{fin}.A); ((\mathit{fin}.B); \mathit{fin}.C) \\
&= \quad \{\text{property } 38(ii) \text{ twice (same steps)}\} \\
&\quad \mathit{fin}.(A;(B;C))
\end{aligned}
$$

□

Property 42 *$true^{set}$ is a neutral element of ";" for both arguments.*

Proof. Let $A \subseteq T$.

$$
\begin{aligned}
&\quad true^{set}; A \\
&= \quad \{\text{definition ";"}\} \\
&\quad \{y,x,a : yx \in true^{set} \wedge a \in A \wedge a.0 = x : ya\} \cup \mathit{inf}.true^{set} \\
&= \quad \{\text{definition } true^{set}\} \\
&\quad \{x,a : a \in A \wedge a.0 = x : a\} \\
&= \quad \{\text{calculus}\} \\
&\quad A
\end{aligned}
$$

Next for the other argument.

$$A; true^{set}$$

$$
\begin{array}{ll}
= & \{\text{definition ";"}\} \\
 & \{a, x, y : ax \in A \wedge y \in true^{set} \wedge y.0 = x : ay\} \cup inf.A \\
= & \{\text{definition } true^{set}\} \\
 & \{a, x : ax \in A \wedge x \in X : ax\} \cup inf.A \\
= & \{\text{calculus}\} \\
 & A
\end{array}
$$

□

Definition 43 *For $A \subseteq T$, we define the powers of A as follows.*

$$A^0 = true^{set}$$
$$A^{n+1} = A; A^n \quad \text{for } n \geq 0.$$

□

We need this notion in the definition of the repetition. However, we need more. As mentioned previously, in an execution of a repetition the body may be executed an infinite number of times. The operational interpretation is an infinite sequence as a limit of finite ones. We therefore continue with studying limits of sequences.

The prefix order on T is a partial order. We are going to define the above limit as a least upper bound of some set. Since T is not a complete lattice, not every set needs to have a least upper bound. However, we show that each ascending chain has one. Consider an ascending chain in T: $c_i, 0 \leq i$. There are two possibilities: there exists an $N \in I\!N$ such that $c_j = c_N$ for $j \geq N$ or no such N exists. In the first case we have $c_N = \sqcup\{i : 0 \leq i : c_i\}$. In the second case we can find a function $f : I\!N \to I\!N$ such that $c_{f.i}, 0 \leq i$ is strictly increasing. Define $d \in T$ elementwise as $d.i = c_{f.i}.i, 0 \leq i$. Notice that d is of infinite length. We show that d is the least upper bound of the chain. Suppose $c_k \not\sqsubseteq d$ for some k. Since $|d| = \infty$, this implies that there exists a j such that $c_k.j \neq d.j$. Hence, $c_k.j \neq c_{f.j}.j$. It follows that c_k and $c_{f.j}$ are incomparable which contradicts the assumption that the c_i form a chain. We conclude that d is an upper bound of the chain. Notice that every upper bound of $\{i : 0 \leq i : c_i\}$ is of infinite length. Assume e is another upper bound, different from d. We have that d and e differ at some index i. It follows that $c_{f.i}$ is not a prefix of e which is in contradiction with the assumption that e is an upper bound of the chain. We conclude that d is the least upper bound of the chain. It follows that, for an ascending chain, the least upper bound is well defined. Notice however that T is not a cpo since the empty string is not an element of T.

Definition 44 *A characterizing chain for an infinite sequence $l \in T$ is an ascending chain $l_i, 0 \leq i$ such that*

(i) $l_i \neq l_{i+1}$,
(ii) $l = \sqcup\{i : 0 \leq i : l_i\}$.

□

Definition 45 *l is a loop point of $A \subseteq T$ if there exists a characterizing chain $l_i, 0 \leq i$ for l such that $l_i \in A^i$. The set of loop points of A is denoted by loop.A.*

□

Property 46 *For $A \subseteq T$ we have $loop.A \cup inf.A = A; loop.A$.*

Proof. The proof is by mutual inclusion. First we prove $\subseteq$. Clearly, $inf.A \subseteq A; loop.A$. Choose $l \in loop.A$ and $l_i, 0 \leq i$ a characterizing chain for l. From the definition of characterizing chain, it follows that all l_i are finite. We can write $l_i, 1 \leq i$ as $l_1 m_i$ since the l_i form an ascending chain and $l = l_1 m$ since l is the least upper bound of the l_i. Since $last.l_1 \in true^{set}$ and $m_i \in \{last.l_1\}; A^i, 1 \leq i$, it follows that $last.l_1, m_i, 1 \leq i$ is a characterizing chain for m in $loop.A$. Since $l_1 \in A$, $\{l_1\}; m \in A; loop.A$.

For the other inclusion, choose $l \in A; loop.A$. Then either $l = am$ for $a(m.0) \in fin.A, m \in loop.A$ or $l \in inf.A$. In the last case we are done. In the first case, we have a characterizing chain for m, $m_i, 0 \leq i$. We construct a new chain, l_i, as follows. $l_0 = a.0$, $l_i = am_{i-1}, 1 \leq i$. Now we have $l_i \in A^i, 0 \leq i$ and $l = \sqcup\{i : 0 \leq i : l_i\}$ hence $l \in loop.A$.

□

Definition 47 *For $A \subseteq T$ we define A^ω as follows.*

$$A^\omega = \bigcup(n : 0 \leq n : A^n) \cup loop.A$$

□

Now we are ready to give the definition of the repetition. We abbreviate **do** $B \rightarrow S$ **od** by DO.

$$DO = (B^{set}; S)^\omega; \neg B^{set}; skip$$

We have to verify that we have defined a proper element of *Prog*. Suppose there exists $x \in X$ such that there is no sequence in DO starting with x. Since sequences starting with x are present in $(B^{set}; S)^\omega$, it follows that they are all finite and terminate in a state satisfying B. Define a characterizing chain as follows. $l_0 = x$, choose $l_{i+1} \in \{l_i\}; S$, arbitrarily. The least upper bound of this chain is a loop point of $(B^{set}; S)$ starting with x. Since it is a loop point, it is in DO. This is a contradiction. It follows that DO is an element of *Prog*.

The above definition of DO is a rather complicated one. It has the advantage however, of defining DO uniquely. An alternative definition of DO starts with the first unfolding. A definition of DO is then obtained by solving

$$DO = \textbf{if } B \rightarrow S; DO \; [] \neg B \rightarrow skip \textbf{ fi}$$

viewed as an equation in sets. This equation itself however, cannot serve as a definition because it does not have a unique solution. This means that we must distinguish among the solutions, which may require a topological approach. This is done for instance in [24] where it is proved that only one solution of the above equation exists that is nonempty and closed with respect to a certain topology, based on a metric. This proof is given only for a language with bounded nondeterminism, i.e., a language in which the number of choices in the alternative statement is

bounded. In our definition we could restrict ourselves to the prefix order on strings and we did not need a metric or a topology. Furthermore, we allow the nondeterminism to be unbounded.

The main reason that we deviate from the usual definition in terms of the first unfolding is that our operational semantics serves as a basis for the definition of different properties. We use the first unfolding to define these properties for the repetition. This gives us some recursive equations. We use the above definition to distinguish among the solutions of these equations.

Since we are going to need that DO is semantically equivalent to its first unfolding, we prove it as a theorem. This is done in the remainder of this section.

Lemma 48 $S^\omega = true^{set} \cup S; S^\omega$

Proof.

$$
\begin{array}{ll}
 & S^\omega \\
= & \quad \{\text{definition}\} \\
 & \bigcup(n : 0 \leq n : S^n) \cup loop.S \\
= & \quad \{\text{property 46: } inf.S \subseteq \bigcup(n : 0 \leq n : S^n)\} \\
 & \bigcup(n : 0 \leq n : S^n) \cup S; loop.S \\
= & \quad \{\text{definition 43, domain split}\} \\
 & true^{set} \cup \bigcup(n : 1 \leq n : S^n) \cup S; loop.S \\
= & \quad \{\text{definition 43}\} \\
 & true^{set} \cup \bigcup(n : 0 \leq n : S; S^n) \cup S; loop.S \\
= & \quad \{\text{property 40}\} \\
 & true^{set} \cup S; \bigcup(n : 0 \leq n : S^n) \cup S; loop.S \\
= & \quad \{\text{property 40}\} \\
 & true^{set} \cup S; (\bigcup(n : 0 \leq n : S^n) \cup loop.S) \\
= & \quad \{\text{definition 47}\} \\
 & true^{set} \cup S; S^\omega
\end{array}
$$

□

Now we are ready for

Theorem 49 $DO =$ **if** $B \rightarrow S; DO \; [] \neg B \rightarrow skip$ **fi**.

Proof.

$$
\begin{array}{ll}
 & \textbf{if } B \rightarrow S; DO \; [] \neg B \rightarrow skip \textbf{ fi} \\
= & \quad \{\text{definition}\} \\
 & B^{set}; S; DO \cup \neg B^{set}; skip \cup false^{set}; abort \\
= & \quad \{false^{set} = \emptyset, \text{ definition } DO\} \\
 & B^{set}; S; (B^{set}; S)^\omega; \neg B^{set}; skip \cup \neg B^{set}; skip \\
= & \quad \{\text{property 42}\}
\end{array}
$$

$$
\begin{array}{ll}
 & B^{set}; S; (B^{set}; S)^{\omega}; \neg B^{set}; skip \cup true^{set}; \neg B^{set}; skip \\
= & \quad \{\text{property 40}\} \\
 & (B^{set}; S; (B^{set}; S)^{\omega} \cup true^{set}); \neg B^{set}; skip \\
= & \quad \{\text{lemma 48}\} \\
 & (B^{set}; S)^{\omega}; \neg B^{set}; skip \\
= & \quad \{\text{definition}\} \\
 & DO
\end{array}
$$

□

This concludes our definition of the operational semantics of our programming language. We use theorem 49 to a great extent when defining properties for the repetition.

5.3 Properties of programs

Now that we have defined the operational semantics of our programming language, we proceed with the introduction of properties of programs. The fact that a program has a certain property means that all its computations share a common characteristic. In other words, a property may be viewed as a set of computations and a program has a property if all its computations are in the property. We add to our notation

$$Prop = \mathcal{P}(T), \text{ the set of properties.}$$

For a particular program and property it is fairly well possible that only part of the computations of the program are in the property. The functions *wlp* and *wp* define the *weakest precondition* for a program such that all its computations have the required property. This can easily be generalized to other properties.

Definition 50 *The weakest precondition is a function* $w : Prog \times Prop \rightarrow P$.

$$w.S.Q.x \;=\; \{x\}; S \subseteq Q$$

□

In order to show that our definition is a generalization of *wlp* and *wp*, we introduce those two functions as a special case.

Definition 51 *The termination functions* $lt, t : P \rightarrow Prop$ *are defined by*

$$
\begin{array}{lll}
lt.p & = & \{s, w : s \in T' \wedge w \in X \wedge (|sw| = \infty \vee p.w) : sw\}, \\
t.p & = & \{s, w : s \in T' \wedge w \in X \wedge |sw| < \infty \wedge p.w : sw\}.
\end{array}
$$

The functions $wlp, wp : Prog \times P \rightarrow P$ *are defined by*

$$
\begin{array}{lll}
wlp.S.p & = & w.S.(lt.p), \\
wp.S.p & = & w.S.(t.p).
\end{array}
$$

□

As an example, we show how the operational semantics serves as a basis to derive *wlp* and *wp* for the constructs in the language. We look at *wlp* for *skip*, $y := e$ and $S;U$ only, since *wlp* and *wp* turn out to be special cases of other properties. From now on we will often use a somewhat different characterization of sequential composition. From property 40 it follows that, for $S, U \subseteq T$,

$$S;U = \bigcup(s, x : sx \in S \wedge x \in X : \{sx\};U).$$

Remark. In the calculations to follow involving weakest preconditions of sequential composition, we always consider sequential composition of elements of *Prog* which are nonempty. Therefore, when using the definition of $S;U$ for programs S and U, we do not have to consider the part *inf.S* explicitly.

For arbitrary program S we have

$$\begin{array}{ll} & wlp.S.p.x \\ = & \quad \{\text{definition } wlp\} \\ & w.S.(lt.p).x \\ = & \quad \{\text{definition } w\} \\ & \{x\};S \subseteq lt.p. \end{array}$$

This yields for *skip*

$$\begin{array}{ll} & wlp.skip.p.x \\ = & \quad \{\text{definition } skip\} \\ & \{x\};\{y : y \in X : yy\} \subseteq lt.p \\ = & \quad \{\text{definition ";"}\} \\ & \{xx\} \subseteq lt.p \\ = & \quad \{\text{definition } lt\} \\ & p.x. \end{array}$$

For $y := e$,

$$\begin{array}{ll} & wlp.(y := e).p.x \\ = & \quad \{\text{definitions assignment and ";"}\} \\ & \{x(x[y/e.x])\} \subseteq lt.p \\ = & \quad \{\text{definition } lt\} \\ & p.(x[y/e.x]). \end{array}$$

In the syntactical representation of predicates, the predicate in the last line is obtained from p by substituting all free occurrences of y in p by e. It is denoted by p_e^y or $p(y := e)$.
The last example is sequential composition.

$$wlp.(S;U).p.x$$

$$
\begin{array}{ll}
= & \{\text{definition } wlp\} \\
& \{x\}; S; U \subseteq lt.p \\
= & \{\text{definitions ";" and } lt\} \\
& \forall(t, w : tw \in \bigsqcup(s, v : sv \in \{x\}; S \wedge v \in X : \{sv\}; U) \wedge w \in X : |tw| = \infty \vee p.w) \\
= & \{tw \in \bigcup(s, v : sv \in \{x\}; S \wedge v \in X : \{sv\}; U) \equiv \\
& \qquad tw = svuw \text{ for } sv \in \{x\}; S, vuw \in \{v\}; U\} \\
& \forall(s, v, u, w : sv \in \{x\}; S \wedge v \in X \wedge vuw \in \{v\}; U \wedge w \in X : |svuw| = \infty \vee p.w) \\
= & \{\text{calculus}\} \\
& \forall(s, v, u, w : sv \in \{x\}; S \wedge v \in X \wedge vuw \in \{v\}; U \wedge w \in X : |sv| = \infty \vee \\
& \qquad |vuw| = \infty \vee p.w) \\
= & \{\text{nesting}\} \\
& \forall(s, v : sv \in \{x\}; S \wedge v \in X : \forall(u, w : vuw \in \{v\}; U \wedge w \in X : |sv| = \infty \vee \\
& \qquad |vuw| = \infty \vee p.w)) \\
= & \{u \text{ and } w \text{ not free in } sv, \text{ renaming the dummy}\} \\
& \forall(s, v : sv \in \{x\}; S \wedge v \in X : |sv| = \infty \vee \\
& \qquad \forall(u, w : uw \in \{v\}; U \wedge w \in X : |uw| = \infty \vee p.w)) \\
= & \{\text{definition } lt\} \\
& \forall(s, v : sv \in \{x\}; S \wedge v \in X : |sv| = \infty \vee \{v\}; U \subseteq lt.p) \\
= & \{\text{definition } wlp\} \\
& \forall(s, v : sv \in \{x\}; S \wedge v \in X : |sv| = \infty \vee wlp.U.p.v) \\
= & \{\text{definition } wlp\} \\
& wlp.S.(wlp.U.p).x
\end{array}
$$

Properties and weakest preconditions can be used in calculations to prove other properties. In order to do so we need calculational rules for both properties and their weakest preconditions. A calculational rule for wlp is, for instance, that $wlp.S$ distributes over universal quantification.

The question arises how to prove facts about properties and weakest preconditions. It seems that there are three possibilities. The first one is to refer to the operational semantics. The fact about properties that we want to prove may be one that holds for all sets of sequences. The drawback of this method has already been pointed out. We have to stick to this operational semantics as a model for our definitions. When defining wlp and wp as functions from programs and predicates to predicates like in [9], we restrict ourselves to the essentials with respect to termination in a certain condition. For that choice of semantics there may be more than one model. We can do something similar for other properties. The second possibility is to perform structural induction over the way a program has been built. This method has the disadvantage that, any time a new language construct is considered, we have to distrust all our knowledge about programs. We have to review all our proofs with respect to this new construct. The third method, the one that we have adopted, is to choose some characteristics of the property. These characteristics or requirements (or healthiness conditions, like in [8]) are proven by structural

induction. They must capture enough of the "core" of the property to prove the things that we want to prove. The validity of the requirements must be checked for every new construct in the language. As an example consider once more the abstract version of *wlp* as a function from programs and predicates to programs. Not every such function is the *wlp* of some program with some predicate. Without referring to the operational semantics we need some restrictions in order to prove facts about *wlp*.

We proceed as follows. We introduce the properties that we are interested in, *ever, leads-to* and *always*, and use the operational interpretation to derive the weakest preconditions for these properties for the constructs in our language. In the next chapters we then explore the properties in more detail. We list the requirements that we impose and prove them without referring to the operational semantics. The weakest preconditions, derived in this chapter, are viewed as definitions in the next chapter.

5.3.1 The property *ever.q*

If a program has the property *ever.q*, q holds at some point in every computation of the program. Following the example in the previous section for *wlp* and *wp*, we define this property in the following way.

Definition 52 *The function ever:* $P \rightarrow$ *Prop is defined by*

$$ever.q = \{s, w, t : swt \in T \wedge |s| < \infty \wedge w \in X \wedge q.w : swt\}$$

or equivalently,

$$ever.q = \{s : s \in T \wedge \exists(k : 0 \leq k < |s| : q.(s.k)) : s\}.$$

□

Following the example of *wlp* and *wp* even further we are tempted to introduce $wev : Prog \times P \rightarrow P$:

$$wev.S.q = w.S.(ever.q).$$

This, however, turns out to be a bad definition in view of sequential composition. We show why.

$$\begin{array}{ll} & wev.(S;U).q.x \\ = & \quad \{\text{definition } wev\} \\ & \{x\};S;U \subseteq ever.q \\ = & \quad \{\text{definition ";"}\} \\ & \forall(t : t \in \bigcup(s, w : sw \in \{x\};S \wedge w \in X : \{sw\};U) : t \in ever.q) \\ = & \quad \{t \in \bigcup(s, w : sw \in \{x\};S \wedge w \in X : \{sw\};U) \equiv \\ & \qquad t = swu, sw \in \{x\};S, wu \in \{w\};U\} \\ & \forall(s, w, u : sw \in \{x\};S \wedge wu \in \{w\};U \wedge w \in X : swu \in ever.q) \\ = & \quad \{\text{convention with respect to infinite sequences, definition } ever\} \\ & \forall(s, w, u : sw \in \{x\};S \wedge wu \in \{w\};U \wedge w \in X : sw \in ever.q \vee \end{array}$$

$$|sw| < \infty \wedge wu \in ever.q)$$

$=$ {nesting, renaming the dummy}

$$\forall(s, w : sw \in \{x\}; S \wedge w \in X : \forall(u : u \in \{w\}; U : sw \in ever.q \vee |sw| < \infty \wedge u \in ever.q))$$

$=$ {calculus}

$$\forall(s, w : sw \in \{x\}; S \wedge w \in X : sw \in ever.q \vee |sw| < \infty \wedge \forall(u : u \in \{w\}; U : u \in ever.q))$$

$=$ {definition}

$$\forall(s, w : sw \in \{x\}; S \wedge w \in X : sw \in ever.q \vee |sw| < \infty \wedge wev.U.q.w)$$

We do not see how to rewrite this formula anymore since universal quantification does not distribute over disjunction. If we, nevertheless, distribute the universal quantification we obtain

$$\forall(s, w : sw \in \{x\}; S \wedge w \in X : |sw| \in ever.q) \ \vee$$
$$\forall(s, w : sw \in \{x\}; S \wedge w \in X : |sw| < \infty \wedge wev.U.q.w)$$

$=$ {definition}

$$wev.S.q.x \vee wp.S.(wev.U.q).x.$$

In terms of our computations, this means that either q becomes *true* in S or S terminates in a state such that q becomes *true* in U. This forbids initial states that "generate" computations of both flavors.

Looking more closely at this example we see that this problem does not originate from the definition of *ever*. The source of the problem is the fact that we look at weakest *pre*conditions. In a recursive definition scheme we want to express the weakest precondition for a new construct in terms of the ones from which it has been built. For $S; U$, this means that we want to express it in terms of S and U. The state in which U is started is a final state of S. This implies that every definition of a weakest precondition for $S; U$ will refer to the final states of S. We give a definition that incorporates this.

Definition 53 *wlev, wev:* $Prog \times P \times P \rightarrow P$*:*

$$wlev.S.q.r = w.S.(ever.q \cup lt.r),$$
$$wev.S.q.r = w.S.(ever.q \cup t.r).$$

□

Similar to *wlp*, the liberal version of *wp*, we also have a liberal version of *wto*, the counterpart of *wlp*. This turns out to be theoretically nicer. Notice that we have an extension of *wp* and *wlp* (choose $q = false$). Notice also that $wev.S.q.false$ is the weakest precondition of *ever.q*. In the remainder of this section we derive *wlev* and *wev* for our language constructs.

$$wlev.skip.q.r.x$$

$=$ {definition}

$$\{xx\} \subseteq ever.q \cup lt.r$$

$=$ {definition}

$$q.x \vee r.x$$

wev.skip.q.r is the same since *skip.x* does not contain any infinite strings.

$$
\begin{array}{ll}
 & \mathit{wlev.abort.q.r.x} \\
= & \quad \{\text{definition}\} \\
 & \{x^\infty\} \subseteq \mathit{ever.q} \cup \mathit{lt.r} \\
= & \quad \{\text{definition}\} \\
 & \mathit{true.x}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{wev.abort.q.r.x} \\
= & \quad \{\text{definition}\} \\
 & \{x^\infty\} \subseteq \mathit{ever.q} \cup \mathit{t.r} \\
= & \quad \{\text{definition}\} \\
 & \mathit{q.x}
\end{array}
$$

In exactly the same way we obtain

$$
\begin{array}{l}
\mathit{wlev.}(y := e).q.r = \mathit{wev.}(y := e).q.r \;=\; q \vee q_e^y \vee r_e^y, \\
\mathit{wlev.}(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).q.r = \forall(i : B_i : \mathit{wlev.}S_i.q.r), \\
\mathit{wev.}(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).q.r = (q \vee \exists(i :: B_i)) \wedge \forall(i : B_i : \mathit{wev.}S_i.q.r).
\end{array}
$$

The only hard constructs are sequential composition and the repetition. Most of the work for sequential composition is similar to what already has been done above.

$$
\begin{array}{ll}
 & \mathit{wlev.}(S;U).q.r.x \\
= & \quad \{\text{definitions of ";" and } \mathit{wlev}\} \\
 & \forall(s,v,u,w : sv \in \{x\}; S \wedge vuw \in \{v\}; U \wedge v,w \in X : svuw \in \mathit{ever.q} \vee \\
 & \qquad |svuw| = \infty \vee r.w) \\
= & \quad \{\text{definition } \mathit{ever}\} \\
 & \forall(s,v,u,w : sv \in \{x\}; S \wedge vuw \in \{v\}; U \wedge v,w \in X : sv \in \mathit{ever.q} \;\vee \\
 & \qquad (|sv| < \infty \wedge vuw \in \mathit{ever.q}) \vee |sv| = \infty \vee |vuw| = \infty \vee r.w) \\
= & \quad \{\text{calculus, nesting}\} \\
 & \forall(s,v : sv \in \{x\}; S \wedge v \in X : \forall(u,w : vuw \in \{v\}; U \wedge w \in X : sv \in \mathit{ever.q} \;\vee \\
 & \qquad (|sv| < \infty \wedge vuw \in \mathit{ever.q}) \vee |sv| = \infty \vee |vuw| = \infty \vee r.w)) \\
= & \quad \{\text{calculus, absorbtion of } |sv| < \infty\} \\
 & \forall(s,v : sv \in \{x\}; S \wedge v \in X : sv \in \mathit{ever.q} \vee |sv| = \infty \;\vee \\
 & \qquad \forall(u,w : vuw \in \{v\}; U : vuw \in \mathit{ever.q} \vee |vuw| = \infty \vee r.w)) \\
= & \quad \{\text{renaming the dummy}\} \\
 & \forall(s,v : sv \in \{x\}; S \wedge v \in X : sv \in \mathit{ever.q} \vee |sv| = \infty \;\vee \\
 & \qquad \forall(u,w : uw \in \{v\}; U : uw \in \mathit{ever.q} \vee |uw| = \infty \vee r.w)) \\
= & \quad \{\text{definition } \mathit{wlev}\}
\end{array}
$$

$$
\begin{array}{ll}
 & \forall(s, v : sv \in \{x\}; S \wedge v \in X : sv \in ever.q \vee |sv| = \infty \vee wlev.U.q.r.v) \\
= & \quad \{\text{definition } wlev\} \\
 & wlev.S.q.(wlev.U.q.r).x
\end{array}
$$

A similar computation for $wev.(S;U).q.r$ yields $wev.(S;U).q.r = wev.S.q.(wev.U.q.r)$.

Finally, we look at DO. From theorem 49 we derive

$$
\begin{array}{ll}
 & wlev.DO.q.r.x \\
= & \quad \{\text{theorem } 49\} \\
 & wlev.(\textbf{if } B \rightarrow S; DO \; []\neg B \rightarrow skip \; \textbf{fi}).q.r.x \\
= & \quad \{wlev \text{ for } IF\} \\
 & (B \wedge wlev.(S;DO).q.r \;\vee\; \neg B \wedge wlev.skip.q.r).x \\
= & \quad \{wlev \text{ for ";" and } skip\} \\
 & (B \wedge wlev.S.q.(wlev.DO.q.r) \;\vee\; \neg B \wedge (q \vee r)).x
\end{array}
$$

Thus $wlev.DO.q.r$ is a solution of the equation in unknown predicate Y:

$$[Y \;\equiv\; B \wedge wlev.S.q.Y \;\vee\; \neg B \wedge (q \vee r)]. \tag{5.1}$$

According to the theorem of Knaster-Tarski (theorem 23), this equation has extreme solutions if $wlev.S.q$ is a monotonic function. Monotonicity of $wlev.S.q$ means that for predicates r and r',

$$[r \Rightarrow r'] \;\Rightarrow\; [wlev.S.q.r \Rightarrow wlev.S.q.r'].$$

This is a direct consequence of the definitions of $wlev$ and lt. Hence equation (5.1) has a strongest and a weakest solution.

Theorem 54 *$wlev.DO.q.r$ is the weakest solution of (5.1).*

Proof. Let y be an arbitrary solution of (5.1). We have to prove that y implies $wlev.DO.q.r$. Let x be such that $y.x$ holds. We have to show that $wlev.DO.q.r.x$ holds. Choose $s \in \{x\}; DO$. It is our obligation to prove that

$$s \in ever.q \vee |s| = \infty \vee r.(last.s),$$

or equivalently,

$$|s| < \infty \wedge s \notin ever.q \Rightarrow r.(last.s).$$

We first prove, by induction on n,

$$|t| < \infty \wedge t \in \{x\}; (B^{set}; S)^n \wedge t \notin ever.q \;\Rightarrow\; y.(last.t). \tag{5.2}$$

The case $n = 0$, gives $t = x$. $y.x$ is given. Assume (5.2) holds, for $n \geq 0$ and consider a finite sequence $t \in \{x\}; (B^{set}; S)^{n+1}$ not in $ever.q$. We have $t = avb, av \in \{x\}; (B^{set}; S)^n, v \in X$. Furthermore,

$$
\begin{array}{ll}
 & |t| < \infty \wedge t \notin ever.q \\
\Rightarrow & \quad \{\text{definition } ever.q\} \\
 & |av| < \infty \wedge av \notin ever.q
\end{array}
$$

hence $y.v$ holds according to the induction hypothesis. Since $vb \in \{v\}; B^{set}; S, B.v$ holds. Since y solves (5.1), we have $wlev.S.q.y.v$ hence, since $b \notin ever.q$, $y.(last.vb)$ holds. Since $last.vb = last.t$, the result follows. This concludes our proof by induction.

Assuming $|s| < \infty$, we can write s as tww with $tw \in \{x\}; (B^{set}; S)^n$ and $w \in X$ for some $n \in \mathbb{N}$. (The doubling of the last element of s stems from the *skip* in the definition of DO.) Since $tww \in DO$, we also have $\neg B.w$. Using (5.2), this gives us

$$\begin{array}{ll} & \neg B.w \wedge tw \notin ever.q \wedge y.w \\ \Rightarrow & \quad \{y \text{ is a solution of (5.1), definition } ever.q\} \\ & \neg q.w \wedge (q \vee r).w \\ = & \quad \{\text{calculus}\} \\ & r.w. \end{array}$$

which is what we had to prove.

□

For $wev.S.q.r$, we obtain the following equation.

$$[Y \equiv B \wedge wev.S.q.Y \vee \neg B \wedge (q \vee r)]. \tag{5.3}$$

Notice that from the definitions of wev and t it follows that $wev.S.q$ is also a monotonic function.

Theorem 55 *$wev.S.q.r$ is the strongest solution of (5.3).*

Proof. We have to prove $[wlev.S.q.r \Rightarrow y]$, for every solution y of (5.3). We prove this by contraposition. Let y be a solution of (5.3) and suppose there exists $x \in X$ such that $wev.DO.q.r.x \wedge \neg y.x$. We show the existence of a characterizing chain $l_k \in \{x\}; DO, 0 \leq k$ of a loop point l such that

$$\forall(k : 0 \leq k : l_k \notin ever.q \wedge \neg y.(last.l_k)). \tag{5.4}$$

We have to show that we can find such an l_k in every $\{x\}; (B^{set}; S)^n$. We prove this by induction. For $n = 0$ we have $l_0 = x$. $\neg y.x$ is given. If $q.x$ holds, we also have $wev.S.q.y.x$ since x is a prefix of all $s \in \{x\}; S$. This is in contradiction with $\neg y.x$ Hence, l_0 satisfies (5.4).

Suppose we have $l_k \in \{x\}; (B^{set}; S)^n$ satisfying (5.4) and $|l_k| < \infty$. Let $last.l_k$ be denoted by w. We prove that $B.w$ holds by contraposition. If $\neg B.w$ holds, $l_k w \in \{x\}; DO$. Since $wev.DO.q.r.x$ holds, $l_k w \in ever.q \vee l_k w \in t.r$. Hence, $r.w$ holds. From the fact that y solves (5.3) we conclude $y.w$ which is a contradiction. It follows that $B.w$ holds. Therefore, $\neg wev.S.q.y.w$ holds hence there exists an $s \in \{l_k\}; B^{set}; S$ such that $s \notin ever.q \wedge s \notin t.p$. Because $wev.DO.q.r.x$ holds, $|s| < \infty$. This s is our l_{k+1}.

The limit of this chain is a loop point $l \notin ever.q$. This contradicts the assumption that $wev.DO.q.r.x$ holds.

□

This ends our introduction of the property $ever.q$. In the next chapter we discuss this property in more detail.

5.3.2 The property *leads-to.p.q*

A program has the property *leads-to.p.q* if for every computation of that program, q holds at or after any point where p holds. Defining a property is now almost a routine.

Definition 56 *leads-to:* $P \times P \rightarrow$ *Prop:*

$$\textit{leads-to.p.q} \;=\; \{s : s \in T \wedge \forall(a, b : s = ab \wedge |a| < \infty : b \in \textit{ever.p} \Rightarrow b \in \textit{ever.q}) : s\}.$$

□

Roughly speaking, if $s \in$ *leads-to.p.q*, we find a q to the right of every p in s. We first give a property of *leads-to* that we need when we deal with the repetition.

Property 57 *For* $a, u \in T', v \in X, |a| < \infty$,

(i) $av \in \textit{leads-to.p.q} \wedge vu \in \textit{leads-to.p.q} \Rightarrow avu \in \textit{leads-to.p.q}$,

(ii) $vu \in \textit{leads-to.p.q} \wedge vu \in \textit{ever.q} \Rightarrow avu \in \textit{leads-to.p.q}$,

(iii) $uv \in \textit{leads-to.p.q} \equiv uvv \in \textit{leads-to.p.q}$.

Proof.

(i)

$avu \in \textit{leads-to.p.q}$

$=$ {definition *leads-to*}

$\forall(x, y : avu = xy \wedge |x| < \infty : y \in \textit{ever.p} \Rightarrow y \in \textit{ever.q})$

$=$ {domain split}

$\forall(x, y : avu = xy \wedge |x| < \infty \wedge x \sqsubseteq av : y \in \textit{ever.p} \Rightarrow y \in \textit{ever.q}) \wedge$

$\forall(x, y : avu = xy \wedge |x| < \infty \wedge a \sqsubseteq x : y \in \textit{ever.p} \Rightarrow y \in \textit{ever.q})$

$=$ {rename dummies, $|a| < \infty$}

$\forall(x, y : av = xy \wedge |x| < \infty : yu \in \textit{ever.p} \Rightarrow yu \in \textit{ever.q}) \wedge$

$\forall(x, y : vu = xy \wedge |x| < \infty : y \in \textit{ever.p} \Rightarrow y \in \textit{ever.q})$

$\Leftarrow$ {definitions *leads-to* and *ever*, $|a| < \infty$}

$\forall(x, y : av = xy \wedge |x| < \infty : y \in \textit{ever.p} \Rightarrow yu \in \textit{ever.q}) \wedge$

$\forall(x, y : av = xy \wedge |x| < \infty : u \in \textit{ever.p} \Rightarrow yu \in \textit{ever.q}) \wedge vu \in \textit{leads-to.p.q}$

$\Leftarrow$ {$|a| < \infty$ hence third term implies second term, definition *ever*}

$\forall(x, y : av = xy \wedge |x| < \infty : y \in \textit{ever.p} \Rightarrow y \in \textit{ever.q}) \wedge vu \in \textit{leads-to.p.q}$

$=$ {definition *leads-to*}

$av \in \textit{leads-to.p.q} \wedge vu \in \textit{leads-to.p.q}$

(ii)

$avu \in \textit{leads-to.p.q}$

$=$ {definition *leads-to*}

$\forall(x, y : avu = xy \wedge |x| < \infty : y \in \textit{ever.p} \Rightarrow y \in \textit{ever.q})$

$=$ {domain split}
$\forall(x,y : avu = xy \wedge |x| < \infty \wedge x \sqsubseteq a : y \in ever.p \Rightarrow y \in ever.q) \wedge$
$\forall(x,y : avu = xy \wedge |x| < \infty \wedge a \sqsubseteq x : y \in ever.p \Rightarrow y \in ever.q)$
$=$ {rename dummies, $|a| < \infty$}
$\forall(x,y : a = xy \wedge |x| < \infty : yvu \in ever.p \Rightarrow yvu \in ever.q) \wedge$
$\forall(x,y : vu = xy \wedge |x| < \infty : y \in ever.p \Rightarrow y \in ever.q)$
$\Leftarrow$ {$|av| < \infty \Rightarrow |y| < \infty$ hence $vu \in ever.q \Rightarrow yvu \in ever.q$, definition *leads-to*}
$vu \in ever.q \wedge vu \in leads\text{-}to.p.q$

(*iii*) follows directly from the definition of *leads-to*.

□

In view of the remarks made about sequential composition in the previous section (with respect to *wev* and *wlev*) we define

Definition 58 *wto, wlto:* $Prog \times P \times P \times P \rightarrow P$*:*

$$\begin{aligned} wlto.S.p.q.r &= w.S.(leads\text{-}to.p.q \cup lt.r), \\ wto.S.p.q.r &= w.S.(leads\text{-}to.p.q \cup t.r). \end{aligned}$$

□

Again, by choosing r to be *false* in the definition of *wto*, we obtain the weakest precondition for *leads-to.p.q*. From the simple constructs, we show how to derive *wlto* for *skip* and $y := e$.

$wlto.skip.p.q.r$
$=$ {definitions *skip* and *wlto*}
$\{x\};\{y : y \in X : yy\} \subseteq leads\text{-}to.p.q \cup lt.r$
$=$ {definition ";"}
$\{xx\} \subseteq leads\text{-}to.p.q \cup lt.r$
$=$ {definition *leads-to* and *lt*}
$(xx \in ever.p \Rightarrow xx \in ever.q) \wedge (x \in ever.p \Rightarrow x \in ever.q) \vee r.x$
$=$ {definition *ever*}
$(p.x \Rightarrow q.x) \vee r.x$

$wlto.(y := e).p.q.r$
$=$ {definition of *wlto*, same steps as above}
$(x(x[y/e.x]) \in ever.p \Rightarrow x(x[y/e.x]) \in ever.q) \wedge$
$(x[y/e.x] \in ever.p \Rightarrow x[y/e.x] \in ever.q) \vee r.(x[y/e.x])$
$=$ {definition *ever*}
$(p.x \Rightarrow (q.x \vee q.(x[y/e.x]))) \wedge (p.(x[y/e.x]) \Rightarrow q.(x[y/e.x])) \vee r.(x[y/e.x])$

In this way we get the following table.

$$
\begin{aligned}
&\mathit{wlto.skip.p.q.r} = \mathit{wto.skip.p.q.r} \;=\; (p \Rightarrow q) \vee r,\\
&\mathit{wlto.abort.p.q.r} = \mathit{true},\\
&\mathit{wto.abort.p.q.r} = p \Rightarrow q,\\
&\mathit{wlto.}(y := e)\mathit{.p.q.r} = \mathit{wto.}(y := e)\mathit{.p.q.r} \;=\; (p \Rightarrow (q \vee q_e^y)) \wedge (p_e^y \Rightarrow q_e^y) \vee r_e^y,\\
&\mathit{wlto.}(\textbf{if}\ [](i :: B_i \rightarrow S_i)\ \textbf{fi})\mathit{.p.q.r} = \forall(i : B_i : \mathit{wlto.}S_i\mathit{.p.q.r}),\\
&\mathit{wto.}(\textbf{if}\ [](i :: B_i \rightarrow S_i)\ \textbf{fi})\mathit{.p.q.r} = ((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : \mathit{wto.}S_i\mathit{.p.q.r}).
\end{aligned}
$$

As always, the difficult constructs are the ones containing sequential composition. We derive only one of *wto* and *wlto* for the semicolon. Since in the previous section we chose the liberal one, we choose the other one this time. We first give two lemmas devoted entirely to the complicated string manipulation required in this derivation.

Lemma 59 *For $s, u \in T', v, w \in X$,*

$$
\begin{aligned}
& \mathit{svuw} \in \mathit{leads\text{-}to.p.q}\\
=\ &\\
& |sv| = \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \vee\\
& |sv| < \infty \wedge \mathit{vuw} \in \mathit{leads\text{-}to.p.q} \wedge (sv \in \mathit{leads\text{-}to.p.q} \vee \mathit{vuw} \in \mathit{ever.q})
\end{aligned}
$$

Proof. The case $|sv| = \infty$ reduces rightaway to $sv \in \mathit{leads\text{-}to.p.q}$. Assume $|sv| < \infty$.

$$
\begin{aligned}
& \mathit{svuw} \in \mathit{leads\text{-}to.p.q}\\
=\ & \quad \{\text{definition}\}\\
& \forall(a, b : ab = \mathit{svuw} \wedge |a| < \infty : b \in \mathit{ever.p} \Rightarrow b \in \mathit{ever.q})\\
=\ & \quad \{ab = \mathit{svuw} \Rightarrow s \sqsubseteq a \vee a \sqsubseteq sv, \text{ domain split, use } |sv| < \infty\}\\
& \forall(a, b : ab = \mathit{svuw} \wedge |a| < \infty \wedge s \sqsubseteq a : b \in \mathit{ever.p} \Rightarrow b \in \mathit{ever.q}) \wedge\\
& \forall(a, b : ab = \mathit{svuw} \wedge a \sqsubseteq sv : b \in \mathit{ever.p} \Rightarrow b \in \mathit{ever.q})\\
=\ & \quad \{\text{rename dummy, twice, } |s| < \infty\}\\
& \forall(a, b : ab = \mathit{vuw} \wedge |a| < \infty : b \in \mathit{ever.p} \Rightarrow b \in \mathit{ever.q}) \wedge\\
& \forall(a, b : ab = sv : \mathit{buw} \in \mathit{ever.p} \Rightarrow \mathit{buw} \in \mathit{ever.q})\\
=\ & \quad \{\text{definition } \mathit{leads\text{-}to}\}\\
& \mathit{vuw} \in \mathit{leads\text{-}to.p.q} \wedge \forall(a, b : ab = sv : \mathit{buw} \in \mathit{ever.p} \Rightarrow \mathit{buw} \in \mathit{ever.q})\\
=\ & \quad \{\text{definition } \mathit{ever}\text{: } \mathit{buw} \in \mathit{ever.p} \equiv (b \in \mathit{ever.p} \vee |b| < \infty \wedge \mathit{uw} \in \mathit{ever.p})\}\\
& \mathit{vuw} \in \mathit{leads\text{-}to.p.q} \wedge \forall(a, b : ab = sv : (b \in \mathit{ever.p} \Rightarrow \mathit{buw} \in \mathit{ever.q}) \wedge\\
& \qquad\qquad (|b| = \infty \vee (\mathit{uw} \in \mathit{ever.p} \Rightarrow \mathit{buw} \in \mathit{ever.q})))\\
=\ & \quad \{s < \infty \text{ hence } |b| < \infty, \text{ distribute quantification}\}\\
& \mathit{vuw} \in \mathit{leads\text{-}to.p.q} \wedge \forall(a, b : ab = sv : b \in \mathit{ever.p} \Rightarrow \mathit{buw} \in \mathit{ever.q}) \wedge\\
& \forall(a, b : ab = sv : \mathit{uw} \in \mathit{ever.p} \Rightarrow \mathit{buw} \in \mathit{ever.q})
\end{aligned}
$$

$=$ {first term implies third term, definition *ever*}

$vuw \in \mathit{leads\text{-}to.p.q} \wedge \forall(a, b : ab = sv : b \in \mathit{ever.p} \Rightarrow (b \in \mathit{ever.q} \vee vuw \in \mathit{ever.q}))$

$=$ {distribute disjunction, definition *leads-to*}

$vuw \in \mathit{leads\text{-}to.p.q} \wedge (sv \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{ever.q})$

□

Lemma 60 *For* $s, u \in T', v, w \in X$,

$$
\begin{array}{l}
|sv| = \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \ \vee \\
|sv| < \infty \wedge vuw \in \mathit{leads\text{-}to.p.q} \wedge (sv \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{ever.q}) \ \vee \\
|svuw| < \infty \wedge r.w \\
= \\
(sv \in \mathit{leads\text{-}to.p.q} \vee |sv| < \infty \wedge (vuw \in \mathit{ever.q} \vee |vuw| < \infty \wedge r.w)) \wedge \\
(|sv| = \infty \vee (vuw \in \mathit{leads\text{-}to.p.q} \vee |vuw| < \infty \wedge r.w)).
\end{array}
$$

Proof.

$$
\begin{array}{l}
(sv \in \mathit{leads\text{-}to.p.q} \vee |sv| < \infty \wedge (vuw \in \mathit{ever.q} \vee |vuw| < \infty \wedge r.w)) \wedge \\
(|sv| = \infty \vee (vuw \in \mathit{leads\text{-}to.p.q} \vee |vuw| < \infty \wedge r.w)) \\
= \quad \{\text{calculus}\} \\
(sv \in \mathit{leads\text{-}to.p.q} \vee |sv| < \infty \wedge (vuw \in \mathit{ever.q} \vee |vuw| < \infty \wedge r.w)) \wedge \\
(|sv| = \infty \vee |sv| < \infty \wedge (vuw \in \mathit{leads\text{-}to.p.q} \vee |vuw| < \infty \wedge r.w)) \\
= \quad \{\text{distribute second term}\} \\
|sv| = \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \ \vee \\
|sv| < \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \wedge vuw \in \mathit{leads\text{-}to.p.q} \ \vee \\
|sv| < \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \wedge |vuw| < \infty \wedge r.w \ \vee \\
|sv| < \infty \wedge vuw \in \mathit{leads\text{-}to.p.q} \wedge vuw \in \mathit{ever.q} \ \vee \\
|sv| < \infty \wedge vuw \in \mathit{leads\text{-}to.p.q} \wedge |vuw| < \infty \wedge r.w \ \vee \\
|sv| < \infty \wedge |vuw| < \infty \wedge r.w \wedge vuw \in \mathit{ever.q} \ \vee \\
|sv| < \infty \wedge |vuw| < \infty \wedge r.w \\
= \quad \{\text{calculus}\} \\
|sv| = \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \ \vee \\
|sv| < \infty \wedge vuw \in \mathit{leads\text{-}to.p.q} \wedge (sv \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{ever.q}) \ \vee \\
|sv| < \infty \wedge |vuw| < \infty \wedge r.w \wedge \\
\qquad (sv \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{ever.q}) \ \vee \\
|sv| < \infty \wedge |vuw| < \infty \wedge r.w \\
= \quad \{\text{one-but-last term implies last term}\} \\
|sv| = \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \ \vee
\end{array}
$$

$$|sv| < \infty \wedge vuw \in \mathit{leads\text{-}to.p.q} \wedge (sv \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{ever.q}) \ \vee$$
$$|sv| < \infty \wedge |vuw| < \infty \wedge r.w$$

□

Now we can give the derivation.

$$\begin{array}{ll}
 & \mathit{wto.}(S;U).p.q.r.x \\
= & \quad \{\text{definition } \mathit{wto}\} \\
 & \{x\}; S; U \subseteq \mathit{leads\text{-}to.p.q} \cup t.r \\
= & \quad \{\text{definition ";"}\} \\
 & \forall(s,v,u,w : sv \in \{x\}; S \wedge vuw \in \{v\}; U \wedge v,w \in X : \\
 & \qquad svuw \in \mathit{lcads\text{-}to.p.q} \vee |svuw| < \infty \wedge r.w) \\
= & \quad \{\text{lemma 59}\} \\
 & \forall(s,v,u,w : sv \in \{x\}; S \wedge vuw \in \{v\}; U \wedge v,w \in X : \\
 & \quad |sv| = \infty \wedge sv \in \mathit{leads\text{-}to.p.q} \ \vee \\
 & \quad |sv| < \infty \wedge vuw \in \mathit{leads\text{-}to.p.q} \wedge (sv \in \mathit{leads\text{-}to.p.q} \vee vuw \in \mathit{ever.q}) \ \vee \\
 & \quad |svuw| < \infty \wedge r.w) \\
= & \quad \{\text{lemma 60}\} \\
 & \forall(s,v,u,w : sv \in \{x\}; S \wedge vuw \in \{v\}; U \wedge v,w \in X : \\
 & \quad (sv \in \mathit{leads\text{-}to.p.q} \vee |sv| < \infty \wedge (vuw \in \mathit{ever.q} \vee |vuw| < \infty \wedge r.w)) \wedge \\
 & \qquad (|sv| = \infty \vee (vuw \in \mathit{leads\text{-}to.p.q} \vee |vuw| < \infty \wedge r.w))) \\
= & \quad \{\text{distribute quantifications, nesting}\} \\
 & \forall(s,v : sv \in \{x\}; S \wedge v \in X : sv \in \mathit{leads\text{-}to.p.q} \vee |sv| < \infty \wedge \\
 & \qquad \forall(u,w : vuw \in \{v\}; U : (vuw \in \mathit{ever.q} \vee |vuw| < \infty \wedge r.w))) \wedge \\
 & \forall(s,v : sv \in \{x\}; S \wedge v \in X : |sv| = \infty \vee \\
 & \qquad \forall(u,w : vuw \in \{v\}; U : vuw \in \mathit{leads\text{-}to.p.q} \vee |vuw| < \infty \wedge r.w)) \\
= & \quad \{\text{definition}\} \\
 & \mathit{wto.S.p.q.}(\mathit{wev.U.q.r}).x \wedge \mathit{wlp.S.}(\mathit{wto.U.p.q.r}).x.
\end{array}$$

A similar calculation for *wlto* yields

$$\mathit{wlto.}(S;U).p.q.r = \mathit{wlto.S.p.q.}(\mathit{wlev.U.q.r}) \wedge \mathit{wlp.S.}(\mathit{wlto.U.p.q.r}).$$

Finally, we inspect the equations for the repetitive construct.

$$\begin{array}{ll}
 & \mathit{wlto.DO.p.q.r.x} \\
= & \quad \{\text{theorem 49}\} \\
 & \mathit{wlto.}(\textbf{if } B \rightarrow S; \mathit{DO} \ [] \neg B \rightarrow \mathit{skip} \ \textbf{fi}).p.q.r.x \\
= & \quad \{\mathit{wlto} \text{ for } \mathit{IF}\}
\end{array}$$

$$
\begin{array}{ll}
 & (B \wedge wlto.(S;DO).p.q.r \ \vee \ \neg B \wedge wlto.skip.p.q.r).x \\
= & \quad \{wlto \text{ for ";" and } skip\} \\
 & (B \wedge wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.(wlto.DO.p.q.r) \ \vee \ \neg B \wedge (\neg p \vee q \vee r)).x
\end{array}
$$

Hence, $wlto.DO.p.q.r$ is a solution of the equation in unknown predicate Y:

$$[Y \ \equiv \ B \wedge wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.Y \ \vee \ \neg B \wedge (\neg p \vee q \vee r)]. \tag{5.5}$$

Again it is trivial to show that $wlto.S.p.q$ is a monotonic function. We have

Theorem 61 *$wlto.DO.p.q.r$ is the weakest solution of (5.5).*

Proof. Let y be an arbitrary solution of (5.5). We have to show that y implies $wlto.DO.p.q.r$. Let x be such that $y.x$ holds. In order to prove the implication, we prove that $wlto.DO.p.q.r.x$ holds. Choose $s \in \{x\};DO$. Our proof obligation becomes

$$|s| < \infty \wedge s \notin leads\text{-}to.p.q \Rightarrow r.(last.s). \tag{5.6}$$

We prove by induction on n

$$
\begin{array}{ll}
 & t \in \{x\};(B^{set};S)^n \wedge |t| < \infty \\
\Rightarrow & \\
 & y.(last.t) \wedge (t \in leads\text{-}to.p.q \vee wlev.DO.q.r.(last.t)).
\end{array} \tag{5.7}
$$

Once we have (5.7), we derive (5.6) as follows. Since $s \in DO$, we can write $s = tww$ for $w \in X$.

$$
\begin{array}{ll}
 & |s| < \infty \wedge s \notin leads\text{-}to.p.q \\
\Rightarrow & \quad \{s \in \{x\};DO, s = tww, \text{ property } 57(iii)\} \\
 & \exists(n : n \in \mathbb{N} : tw \in \{x\};(B^{set};S)^n) \wedge \neg B.w \wedge tw \notin leads\text{-}to.p.q \wedge |tw| < \infty \\
\Rightarrow & \quad \{(5.7), tw \notin leads\text{-}to.p.q \wedge |tw| < \infty \Rightarrow \neg q.w\} \\
 & y.w \wedge wlev.DO.q.r.w \wedge \neg B.w \wedge \neg q.w \\
\Rightarrow & \quad \{wlev.DO.q.r \text{ solves (5.1)}\} \\
 & \neg q.w \wedge (q.w \vee r.w) \\
\Rightarrow & \quad \{\text{calculus}\} \\
 & r.w \\
= & \quad \{w = last.s\} \\
 & r.(last.s)
\end{array}
$$

We prove (5.7). If $n = 0$, $t = x$ and $y.x$ is given. We perform case analysis between $\neg B.x$ and $B.x$. First, $\neg B.x$.

$$
\begin{array}{ll}
 & y.x \\
= & \quad \{y \text{ solves (5.5)}, \neg B.x \text{ holds}\} \\
 & (\neg p \vee q \vee r).x \\
= & \quad \{\text{calculus}\} \\
 & (\neg p \vee q).x \vee (q \vee r).x \\
\Rightarrow & \quad \{\text{definition } leads\text{-}to, wlev.DO.q.r \text{ solves (5.1)}, \neg B.x \text{ holds}\} \\
 & x \in leads\text{-}to.p.q \vee wlev.DO.q.r.x
\end{array}
$$

The case $B.x$ is more complicated.

$$
\begin{array}{ll}
 & y.x \\
\Rightarrow & \quad \{y \text{ solves (5.5)}, B.x \text{ holds}\} \\
 & wlto.S.p.q.(wlev.DO.q.r).x \\
= & \quad \{\text{definition}\} \\
 & \forall(s : s \in \{x\}; S : |s| = \infty \vee s \in leads\text{-}to.p.q \vee |s| < \infty \wedge wlev.DO.q.r.(last.s)) \\
\Rightarrow & \quad \{x \sqsubseteq s \wedge s \in leads\text{-}to.p.q \Rightarrow x \in leads\text{-}to.p.q \vee s \in ever.q\} \\
 & \forall(s : s \in \{x\}; S : |s| = \infty \vee x \in leads\text{-}to.p.q \vee s \in ever.q \vee \\
 & \qquad |s| < \infty \wedge wlev.DO.q.r.(last.s)) \\
= & \quad \{\text{calculus}\} \\
 & x \in leads\text{-}to.p.q \vee \forall(s : s \in \{x\}; S : |s| = \infty \vee s \in ever.q \vee \\
 & \qquad |s| < \infty \wedge wlev.DO.q.r.(last.s)) \\
= & \quad \{\text{definition } wlev\} \\
 & x \in leads\text{-}to.p.q \vee wlev.S.q.(wlev.DO.q.r).x \\
= & \quad \{B.x \text{ holds}, wlev.DO.q.r \text{ solves (5.1)}\} \\
 & x \in leads\text{-}to.p.q \vee wlev.DO.q.r.x
\end{array}
$$

This was the base case of the induction. Suppose we have (5.7) for $n \geq 0$. Choose $t \in \{x\}; (B^{set}; S)^{n+1}, |t| < \infty$. t can be written as avu, $av \in \{x\}; (B^{set}; S)^n, v \in X$. Notice, $|av| < \infty$. Since av is a prefix of t we also have $B.v$. We collect what we get from the induction hypothesis for av.

$$
\begin{array}{ll}
 & y.v \wedge (av \in leads\text{-}to.p.q \vee wlev.DO.q.r.v) \\
\Rightarrow & \quad \{y \text{ solves (5.5)}, |vu| < \infty, \text{ use (5.1) and } B.v\} \\
 & y.(last.vu) \wedge (vu \in leads\text{-}to.p.q \vee wlev.DO.q.r.(last.vu)) \wedge \\
 & \qquad (av \in leads\text{-}to.p.q \vee wlev.S.q.(wlev.DO.q.r).v) \\
\Rightarrow & \quad \{\text{definition } wlev, |vu| < \infty\} \\
 & y.(last.vu) \wedge (vu \in leads\text{-}to.p.q \vee wlev.DO.q.r.(last.vu)) \wedge \\
 & \qquad (av \in leads\text{-}to.p.q \vee vu \in ever.q \vee wlev.DO.q.r.(last.vu)) \\
= & \quad \{\text{distribute second term}\} \\
 & y.(last.vu) \wedge \\
 & (vu \in leads\text{-}to.p.q \wedge av \in leads\text{-}to.p.q \vee \\
 & vu \in leads\text{-}to.p.q \wedge vu \in ever.q \vee \\
 & vu \in leads\text{-}to.p.q \wedge wlev.DO.q.r.(last.vu) \vee \\
 & wlev.DO.q.r.(last.vu) \wedge av \in leads\text{-}to.p.q \vee \\
 & wlev.DO.q.r.(last.vu) \wedge vu \in ever.q \vee \\
 & wlev.DO.q.r.(last.vu)) \\
\Rightarrow & \quad \{\text{property } 57(i),(ii), \text{calculus}\}
\end{array}
$$

$$y.(last.vu) \wedge (avu \in leads\text{-}to.p.q \vee wlev.DO.q.r.(last.vu))$$

This means that we are done with the induction and, hence, with the proof of this theorem. □

The equation for *wto* is obtained in a similar way using theorem 49.

$$[Y \equiv B \wedge wto.S.p.q.(wev.DO.q.r) \wedge wlp.S.Y \vee \neg B \wedge (\neg p \vee q \vee r)]. \tag{5.8}$$

wto.DO.p.q is also a monotonic function on predicates. The following theorem may be a bit surprising.

Theorem 62 *wto.DO.p.q.r is the weakest solution of (5.8).*

Proof. We follow similar steps as in the previous proof. Let y be an arbitrary solution of (5.8). We have to show that y implies *wlto.DO.p.q.r*. Let x be such that $y.x$ holds and $s \in \{x\}; DO$. We must show that *wto.DO.p.q.r.x* holds, which reduces to

$$s \notin leads\text{-}to.p.q \Rightarrow |s| < \infty \wedge r.(last.s). \tag{5.9}$$

Unfortunately, we now have to distinguish between two cases: s is a loop point and s is not a loop point. We first prove (5.9) for the case that s is not a loop point. We prove by induction on n that, for all $t \in \{x\}; (B^{set}; S)^n$ we have

$$\begin{array}{l} |t| = \infty \wedge t \in leads\text{-}to.p.q \vee \\ |t| < \infty \wedge y.(last.t) \wedge (t \in leads\text{-}to.p.q \vee wev.DO.q.r.(last.t)). \end{array} \tag{5.10}$$

Again, once we have (5.10), we derive (5.9) straightforwardly. Notice that s can be written as tww with $w \in X$ regardless whether s is finite or infinite.

$$\begin{array}{ll} & s \notin leads\text{-}to.p.q \\ \Rightarrow & \{s \in \{x\}; DO,\ s \text{ not a loop point},\ s = tww,\ \text{property } 57(iii)\} \\ & \exists(n : n \in \mathbb{N} : tw \in \{x\}; (B^{set}; S)^n) \wedge (|tw| = \infty \vee \neg B.w) \wedge tw \notin leads\text{-}to.p.q \\ \Rightarrow & \{(5.10),\ tw \notin leads\text{-}to.p.q \Rightarrow \neg q.w \wedge |tw| < \infty\} \\ & |tw| < \infty \wedge y.w \wedge wev.DO.q.r.w \wedge \neg B.w \wedge \neg q.w \\ \Rightarrow & \{wev.DO.q.r \text{ solves } (5.3)\} \\ & |tw| < \infty \wedge \neg q.w \wedge (q \vee r).w \\ \Rightarrow & \{\text{calculus},\ tww = s\} \\ & |s| < \infty \wedge r.(last.s) \end{array}$$

We prove (5.10). If $n = 0$, $t = x$, $|x| < \infty$ and $y.x$ is given. First assume $\neg B.x$.

$$\begin{array}{ll} & y.x \\ = & \{y \text{ solves } (5.8),\ \neg B.x \text{ holds}\} \\ & (\neg p \vee q \vee r).x \\ = & \{\text{calculus}\} \\ & (\neg p \vee q).x \vee (q \vee r).x \\ \Rightarrow & \{\text{definition } leads\text{-}to,\ wev.DO.q.r \text{ solves } (5.3),\ \neg B.x \text{ holds}\} \\ & x \in leads\text{-}to.p.q \vee wev.DO.q.r.x \end{array}$$

Next the case $B.x$.

$$
\begin{array}{ll}
 & y.x \\
\Rightarrow & \quad \{y \text{ solves (5.8)}, B.x \text{ holds}\} \\
 & wto.S.p.q.(wev.DO.q.r).x \\
= & \quad \{\text{definition}\} \\
 & \forall(s : s \in \{x\}; S : s \in leads\text{-}to.p.q \vee |s| < \infty \wedge wev.DO.q.r.(last.s)) \\
\Rightarrow & \quad \{x \sqsubseteq s \wedge s \in leads\text{-}to.p.q \Rightarrow x \in leads\text{-}to.p.q \vee s \in ever.q\} \\
 & \forall(s : s \in \{x\}; S : x \in leads\text{-}to.p.q \vee s \in ever.q \vee |s| < \infty \wedge wev.DO.q.r.(last.s)) \\
= & \quad \{\text{calculus}\} \\
 & x \in leads\text{-}to.p.q \vee \forall(s : s \in \{x\}; S : s \in ever.q \vee |s| < \infty \wedge wev.DO.q.r.(last.s)) \\
= & \quad \{\text{definition } wev\} \\
 & x \in leads\text{-}to.p.q \vee wev.S.q.(wev.DO.q.r).x \\
= & \quad \{B.x \text{ holds}, wev.DO.q.r \text{ solves (5.3)}\} \\
 & x \in leads\text{-}to.p.q \vee wev.DO.q.r.x
\end{array}
$$

This proves the base case. Suppose we have (5.10) for $n \geq 0$. Choose $t \in \{x\}; (B^{set}; S)^{n+1}, |t| < \infty$. t can be written as avu, $av \in \{x\}; (B^{set}; S)^n, v \in X$. From this, it follows that $|av| < \infty$. Since av is a prefix of t we also have $B.v$. We apply the induction hypothesis on av.

$$
\begin{array}{ll}
 & y.v \wedge (av \in leads\text{-}to.p.q \vee wev.DO.q.r.v) \\
\Rightarrow & \quad \{y \text{ solves (5.8)}, \text{use (5.3) and } B.v\} \\
 & (|vu| = \infty \vee |vu| < \infty \wedge y.(last.vu)) \wedge \\
 & (vu \in leads\text{-}to.p.q \vee |vu| < \infty \wedge wev.DO.q.r.(last.vu)) \wedge \\
 & (av \in leads\text{-}to.p.q \vee wev.S.q.(wev.DO.q.r).v) \\
\Rightarrow & \quad \{\text{definition } wev\} \\
 & (|vu| = \infty \vee |vu| < \infty \wedge y.(last.vu)) \wedge \\
 & (vu \in leads\text{-}to.p.q \vee |vu| < \infty \wedge wev.DO.q.r.(last.vu)) \wedge \\
 & (av \in leads\text{-}to.p.q \vee vu \in ever.q \vee |vu| < \infty \wedge wev.DO.q.r.(last.vu)) \\
= & \quad \{\text{distribute disjunctions}\} \\
 & |vu| = \infty \wedge vu \in leads\text{-}to.p.q \wedge av \in leads\text{-}to.p.q \vee \\
 & |vu| = \infty \wedge vu \in leads\text{-}to.p.q \wedge vu \in ever.q \vee \\
 & |vu| < \infty \wedge y.(last.vu) \wedge vu \in leads\text{-}to.p.q \wedge av \in leads\text{-}to.p.q \vee \\
 & |vu| < \infty \wedge y.(last.vu) \wedge vu \in leads\text{-}to.p.q \wedge vu \in ever.q \vee \\
 & |vu| < \infty \wedge y.(last.vu) \wedge vu \in leads\text{-}to.p.q \wedge wev.DO.q.r.(last.vu) \vee \\
 & |vu| < \infty \wedge y.(last.vu) \wedge wev.DO.q.r.(last.vu) \wedge av \in leads\text{-}to.p.q \vee \\
 & |vu| < \infty \wedge y.(last.vu) \wedge wev.DO.q.r.(last.vu) \wedge vu \in ever.q \vee \\
 & |vu| < \infty \wedge y.(last.vu) \wedge wev.DO.q.r.(last.vu) \\
\Rightarrow & \quad \{|a| < \infty, \text{property } 57(i),(ii), \text{calculus}\}
\end{array}
$$

$$
\begin{array}{ll}
 & |vu| = \infty \wedge avu \in \mathit{leads\text{-}to.p.q} \vee \\
 & |vu| < \infty \wedge y.(\mathit{last}.vu) \wedge avu \in \mathit{leads\text{-}to.p.q} \vee \\
 & |vu| < \infty \wedge y.(\mathit{last}.vu) \wedge \mathit{wev.DO.q.r.}(\mathit{last}.vu) \\
= & \quad \{\text{calculus}\} \\
 & (|vu| = \infty \wedge avu \in \mathit{leads\text{-}to.p.q}) \vee \\
 & (|vu| < \infty \wedge y.(\mathit{last}.vu) \wedge (avu \in \mathit{leads\text{-}to.p.q} \vee \mathit{wev.DO.q.r.}(\mathit{last}.vu))) \\
\Rightarrow & \quad \{|a| < \infty\} \\
 & (|avu| = \infty \wedge avu \in \mathit{leads\text{-}to.p.q}) \vee \\
 & (|avu| < \infty \wedge y.(\mathit{last}.avu) \wedge (avu \subset \mathit{lcads\text{-}to.p.q} \vee \mathit{wev.DO.q.r.}(\mathit{last}.avu)))
\end{array}
$$

This completes our proof by induction and, hence, the proof of (5.9) for the case that s is not a loop point.

Assume that s is a loop point. Let $l_k, 0 \leq k$ be a characterizing chain for s. Assume $s \notin \mathit{leads\text{-}to.p.q}$. Then there exist $a, b \in T, |a| < \infty$ such that $s = ab$ and $b \in \mathit{ever.p} \wedge b \notin \mathit{ever.q}$. We may choose this b to be maximal. From the definition (45) we have $l_k \sqsubset l_{k+1} \sqsubset s$ for all k. Since the chain is strictly increasing, there will be $n \in \mathbb{N}$ such that $a \sqsubseteq l_n$. For $k > n$, we can write l_k as am_k. Since b is maximal, $m_k \notin \mathit{ever.q}$ for $k > n$. Since $b \in \mathit{ever.p}$, there exists $h \geq n$ such that $m_h \in \mathit{ever.p}$. We have: $l_k \notin \mathit{leads\text{-}to.p.q}$ for $k \geq h$ (from the above), $B.(\mathit{last}.l_h)$ (since the l_k form a characterizing chain) and $y.(\mathit{last}.l_h)$ (by (5.10)). Since $l_{h+1} \in \{l_h\}; B^{\mathit{set}}; S$, we conclude from (5.10) that $\mathit{wev.DO.q.r.}(\mathit{last}.l_h)$ holds. We can write l_k, $k \geq h$ as $l_h c_k$ and s as $l_h t$; we use this to construct a characterizing chain d_k for a loop point in $\{\mathit{last}.l_h\}; DO$ that is not in $\mathit{ever.q}$. Define $d_k = (\mathit{last}.l_h)c_{h+k}$ for $0 \leq k$. The limit of this chain is $(\mathit{last}.l_h)t$. By construction, $(\mathit{last}.l_h)t \notin \mathit{ever.q}$. This contradicts the fact that $\mathit{wev.DO.q.r.}(\mathit{last}.l_h)$ holds. It follows that the decomposition of $s = ab$ is not possible. We conclude $s \in \mathit{leads\text{-}to.p.q}$.

□

This ends our introduction of the property *leads-to.p.q*. In one of the chapters to follow, we investigate this property in more detail.

5.3.3 The property *always.p*

The last property that we consider is the property *always.p*. If *always.p* is a property of a program, p holds in every state of every computation of the program. As before we define

Definition 63 *The function always:* $P \rightarrow$ *Prop,*

$$\mathit{always.p} = \{s : s \in T \wedge \forall(i : 0 \leq i < |s| : p.(s.i)) : s\}.$$

□

We can phrase this a bit differently: *always.p* consists of those sequences s such that p is never *false* in s. Hence, $s \in \mathit{always.p} \equiv s \notin \mathit{ever.}\neg p$. How can we easily describe that $s \notin \mathit{ever.p}$? It turns out that we can use the property *leads-to* by the observation that "p never becomes *true* in s" is equivalent to "p ever becoming *true* in s is *false*".

$$s \notin \mathit{ever.}\neg p$$

$$
\begin{array}{ll}
= & \{\text{calculus}\} \\
& s \in ever.\neg p \Rightarrow false \\
= & \{\text{definition } ever\} \\
& \exists(a,w,b : |a| < \infty \wedge x \in X \wedge \neg p.x : s = awb) \Rightarrow false \\
= & \{\text{definition } ever, \text{ renaming dummy}\} \\
& \exists(a,b : |a| < \infty \wedge b \in ever.\neg p : s = ab) \Rightarrow false \\
= & \{\text{trading}\} \\
& \exists(a,b : |a| < \infty \wedge s = ab : b \in ever.\neg p) \Rightarrow false \\
= & \{\text{calculus}\} \\
& \forall(a,b : |a| < \infty \wedge s = ab : b \notin ever.\neg p) \\
= & \{ever.false = \emptyset\} \\
& \forall(a,b : |a| < \infty \wedge s = ab : b \in ever.\neg p \Rightarrow b \in ever.false) \\
= & \{\text{definition}\} \\
& s \in leads\text{-}to.\neg p.false
\end{array}
$$

It turns out that the property *always.p* is a special case of one of the properties that we already know. If we denote the weakest liberal always by *wlalw* and the weakest always by *walw* we obtain the following weakest preconditions.

$$
\begin{array}{ll}
wlalw.skip.p = p & walw.skip.p = p \\
wlalw.abort.p = true & walw.abort.p = p \\
wlalw.(y := e).p = p \wedge p_e^y & walw.(y := e).p = p \wedge p_e^y
\end{array}
$$

$$
\begin{array}{l}
wlalw.(S;U).p = wlalw.S.p \wedge wlp.S.(wlalw.U.p) \\
walw.(S;U).p = walw.S.p \wedge wlp.S.(walw.U.p) \\
wlalw.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).p = \forall(i : B_i : wlalw.S_i.p) \\
walw.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).p = (p \vee \exists(i :: B_i)) \wedge \forall(i : B_i : walw.S_i.p)
\end{array}
$$

$wlalw.(\textbf{do } B \rightarrow S \textbf{ od}).p$ is the weakest solution of equation

$$[Y \;=\; B \wedge wlalw.S.p \wedge wlp.S.Y \;\vee\; \neg B \wedge p]$$

in unknown Y. $walw.(DO).p$ is the weakest solution of

$$[Y \;\equiv\; B \wedge walw.S.p \wedge wlp.S.Y \;\vee\; \neg B \wedge p].$$

Because *always* is a special case of *leads-to*, we do not pay attention to this property anymore. We note however that *leads-to* is a more complicated property than *always* which means that it may pay to study *always* in isolation.

5.4 Determinism and nondeterminism

In this section we pay some attention to the notion of determinism. In [9] the following concise definition of determinism has been given. Program S is deterministic if

$$[wp.S.p = \neg wlp.S.(\neg p)], \text{ for all } p.$$

Loosely phrased, this definition expresses that there is only one possible outcome for the program S. This definition does not suffice anymore since it defines determinism only with respect to a certain property. Since we have generalized the concept of a property of a program, we have to generalize the concept of determinism as well. Consider the following program.

if $true \rightarrow x := 3; y := 4$
[] $true \rightarrow y := 4; x := 3$
fi

According to the above definition of determinism, this program is deterministic. However, we can use *wlev* or *wev* to discriminate between the two alternatives. For instance, if the program is started in a state satisfying $y = 5$, in the first alternative there is a state satisfying $y = 5 \wedge x = 3$ which is not present in the second alternative. We come back to this in the next chapter when we have more tools to compute *wlev* and *wev*.

It appears that we have two choices. We can define program S being deterministic as "S has only one thread of execution on every initial state". In other words, $|\{x\}; S| = 1$ for all $x \in X$. This might capture our intuition with respect to determinism but it is also very limiting. In the above example, as long as we are interested only in final states, both possibilities are indistinguishable. Therefore, we define determinism a little bit more liberally. We define it with respect to a property.

Definition 64 *Program S is deterministic with respect to property $Q \in Prop$ if*

$$\{x\}; S \subseteq Q \ \vee \ \{x\}; S \subseteq (T \setminus Q)$$

or equivalently,

$$\neg(\{x\}; S \subseteq Q) \ \equiv \ \{x\}; S \subseteq (T \setminus Q)$$

for all $x \in X$. S is deterministic with respect to a class of properties if it is deterministic with respect to all properties in the class.

□

Notice that the original definition defines determinism with respect to a class. Using the definition of weakest precondition we derive that program S is deterministic with respect to property Q if

$$[\neg w.S.Q \ \equiv \ w.S.(T \setminus Q)].$$

We look at what this definition yields for the properties that we have until now. We first look at $ever.q \cup lt.r$ and we calculate the right-hand side of the above definition.

$$\begin{array}{ll} & T \setminus (ever.q \cup lt.r) \\ = & \quad \{\text{definitions } ever, lt\} \\ & \{t : t \in T \wedge \forall(i : 0 \leq i < |t| : \neg q.(t.i)) \wedge |t| < \infty \wedge \neg r.(last.t) : t\} \\ = & \quad \{\text{calculus}\} \\ & \{t : t \in T \wedge \forall(i : 0 \leq i < |t| : \neg q.(t.i)) : t\} \cap \{t : t \in T \wedge |t| < \infty \wedge \neg r.(last.t) : t\} \\ = & \quad \{\text{definitions } always, t\} \\ & always.\neg q \ \cap \ t.\neg r \end{array}$$

Hence, program S is deterministic with respect to $ever.q \cup lt.r$ if

$$[\neg wlev.S.q.r \;\equiv\; wto.S.q.false.false \wedge wp.S.\neg r].$$

A similar calculation yields: S is deterministic with respect to $ever.q \cup t.r$ if

$$[\neg wev.S.q.r \;\equiv\; wto.S.q.false.false \wedge wlp.S.\neg r].$$

We extend this naturally to the class of properties associated with $ever.q \cup lt.r$ by universal quantification over q and r. We call determinism with respect to this class, determinism with respect to *wlev* and *wev*. In the next chapter, we give an example when we investigate the functions *wlev* and *wev* in more detail. However, some remarks must be made. Firstly, if we substitute *false* for q, this characterization reduces to the original characterization of determinism. Secondly, the reason that we could write down the above formula is that we were very fortunate to be able to characterize the complement of *ever.q*. It is not always possible to characterize this complement so easily as we shall see for *wlto* and *wto*. If we really want to investigate what being deterministic means for those two functions, we have to define the complement of *leads-to.p.q* by introducing its weakest precondition. And thirdly, determinism with respect to *wlev* is something else than determinism with respect to *wev*, which is a little surprising.

We calculate the complement of $leads\text{-}to.p.q \cup lt.r$.

$$\begin{array}{ll} & T \setminus (leads\text{-}to.p.q \cup lt.r) \\ = & \quad \{\text{definitions } leads\text{-}to \text{ and } lt\} \\ & \{t : t \in T \wedge \exists(a,b : |a| < \infty \wedge t = ab : b \in ever.p \wedge b \notin ever.q) \wedge \\ & \qquad |t| < \infty \wedge \neg r.(last.t) : t\} \\ = & \quad \{\text{calculus}\} \\ & \{t : t \in T \wedge \exists(a,b : |a| < \infty \wedge t = ab : b \in ever.p \wedge b \notin ever.q) : t\} \cap \\ & \qquad \{t : t \in T : |t| < \infty \wedge \neg r.(last.t) : t\} \end{array}$$

We do not see how to characterize the first set. A guess that comes very close is $leads\text{-}to.q.p \cap ever.p$ but this also contains strings in which for the last state satisfying q, p holds as well. We therefore do not investigate determinism with respect to *leads-to.p.q*.

Chapter 6

The property ever.q

6.1 Introduction

In the preceding chapter, we have seen how to use the operational semantics of our programming language to introduce properties of programs. Properties were defined for programs by introducing their weakest preconditions and a program has a property if it is started in a state satisfying this precondition. The operational semantics was used to derive these weakest preconditions of some properties for each of the constructs in the programming language. One of the properties we introduced was *ever.q*, defined by two functions (i.e., weakest preconditions): *wlev* and *wev*. In this chapter we explore these functions in greater detail.

As mentioned in the introduction of chapter five, the operational semantics may be overspecific with respect to the properties we are interested in. In this particular case, we are interested only in the properties *ever* and *leads-to*. We take the definitions of the corresponding weakest preconditions to be the semantics of our programming language (and leave the operational semantics for what it is). The essential characteristics of these properties will be captured in a number of requirements (or healthiness conditions as in [8]). In our proofs we will not refer to the operational semantics anymore. This makes the task of an implementer of the language easier.

In the next section we give the requirements that distinguish between functions that can and those that cannot be the *wlev* and the *wev* of a statement. We use these requirements to prove some theorems about the two functions. The last section contains, for every construct, the proofs that it satisfies these requirements. But first we list the definitions given for each construct.

$$
\begin{array}{ll}
wlev.skip.q.r = q \vee r & wev.skip.q.r = q \vee r \\
wlev.abort.q.r = true & wev.abort.q.r = q \\
wlev.(y := e).q.r = q \vee q_e^y \vee r_e^y & wev.(y := e).q.r = q \vee q_e^x \vee r_e^y \\
wlev.(S;U).q.r = wlev.S.q.(wlev.U.q.r) & wev.(S;U).q.r = wev.S.q.(wev.U.q.r)
\end{array}
$$

$$
\begin{array}{l}
wlev.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).q.r = \forall(i : B_i : wlev.S_i.q.r) \\
wev.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).q.r \ = (q \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wev.S_i.q.r)
\end{array}
$$

The definitions for **do** $B \rightarrow S$ **od** were a bit different. We had two equations in unknown

predicate Y.

$$[Y \equiv B \wedge wlev.S.q.Y \ \vee \ \neg B \wedge (q \vee r)]$$
$$[Y \equiv B \wedge wev.S.q.Y \ \vee \ \neg B \wedge (q \vee r)]$$

We write these equations a little differently.

$$[Y \equiv (\neg B \vee wlev.S.q.Y) \ \wedge \ (B \vee q \vee r)] \tag{6.1}$$
$$[Y \equiv (\neg B \vee wev.S.q.Y) \ \wedge \ (B \vee q \vee r)] \tag{6.2}$$

$wlev.DO.q.r$ is the weakest solution of (6.1) and $wev.DO.q.r$ is the strongest solution of (6.2). The right-hand sides of both equations, are functions of three predicates. We call these functions, F and G respectively.

6.2 Requirements for *wlev* and *wev*

In the previous chapter we defined *wlev* and *wev* using the functions *ever.q*, *lt.r* and *t.r*. In this way we can imagine the computations (i.e., the set T) to be partitioned into four classes.

"ever q":	all computations during which q holds at some point.
"never q and eternal":	all nonterminating computations during which q never holds.
"never q and finally r":	all computations during which q never holds and that terminate in r.
"never q and finally $\neg r$":	all computations during which q never holds and that terminate in $\neg r$.

With respect to this classification we can restate the meaning of *wlev* and *wev*. $wlev.S.q.r$ holds exactly in those initial states for which no computation of S belongs to the class "never q and finally $\neg r$". $wev.S.q.r$ holds exactly in those initial states for which no computation of S belongs to the class "never q and finally $\neg r$" or to the class "never q and eternal".

Both $wlev.S$ and $wev.S$ are functions from pairs of predicates to predicates. However, not every such function is a $wlev.S$ or $wev.S$ for some program S. We use the operational interpretation to obtain some requirements that restrict the class of functions.

If we choose for q the constant predicate *false*, the class "ever q" will be empty. Moreover, the partitioning of the class "never q" is the same as the one underlying the definition of *wp* and *wlp* in [9]. The definitions of *wlev* and *wev* reduce to the definitions of those two functions. This gives us our first requirement.

$$e0: \ [wlev.S.false.r \equiv wlp.S.r]$$
$$e0': \ [wev.S.false.r \equiv wp.S.r]$$

Since $wlp.S$ is universally conjunctive, we have that $wlev.S.false$ is universally conjunctive. According to our classification, we can also require the conjunctivity for arbitrary q.

$e1$: $wlev.S.q$ is universally conjunctive

wlp and *wp* are related by the nice formula $[wp.S.r \equiv wp.S.true \wedge wlp.S.r]$. From our classification we obtain: *wev.S.q.true* holds exactly in those initial states for which no computation of S belongs to the class "never q and eternal". This means that we indeed have a similar requirement for *wlev* and *wev*.

$e2: \ [wev.S.q.r \equiv wev.S.q.true \ \wedge \ wlev.S.q.r]$

Because of $e1$, $[wlev.S.q.true]$ which is consistent with $e2$. From $e1$ and $e2$ we derive

$e1': \ wev.S.q$ is positively conjunctive.

We have another extreme choice for r, *false*. *wev.S.q.false* holds exactly in those initial states such that no computation of S is in "never q". Hence, *wev.S.q.false* is the weakest precondition for the property that we started with: *ever.q*.

We already investigated the extreme choice *false* for q. Choosing q to be *true* leaves the class "never q" empty and, hence, we may require both $[wlev.S.true.r]$ and $[wev.S.true.r]$. We require something that is even stronger since if S is started in a state satisfying q, all computations of S will be in "ever q".

$e3': \ [q \Rightarrow wev.S.q.r]$

The corresponding requirement for *wlev* follows from $e2$ and $e3'$.

$e3: \ [q \rightarrow wlev.S.q.r]$

Suppose that we have two predicates, q and q' such that q is stronger than q'. Since q' holds in every state in which q holds, the class "ever q" is included in the class "ever q'".

$e4: \ [q \Rightarrow q'] \Rightarrow [wlev.S.q.r \Rightarrow wlev.S.q'.r]$
$e4': \ [q \Rightarrow q'] \Rightarrow [wev.S.q.r \Rightarrow wev.S.q'.r]$

We already had a requirement that refers to the state in which a computation is started. Our final requirement concerns the state upon termination. Consider a computation in the class "never q and finally $\neg r$". Since q never holds, it is also the case that q does not hold upon termination.

$e5: \ [wlev.S.q.r \equiv wlev.S.q.(q \vee r)]$

Combining $e2$ and $e5$ yields the counterpart for *wev*.

$e5': \ [wev.S.q.r \equiv wev.S.q.(q \vee r)]$

We end up with a lot of requirements for *wlev* and *wev*, viz. six, some of which are formulated for both *wlev* and *wev*. We show that we cannot drop one, i.e., we show that none follows from the others. We do so by choosing for *wlev* an arbitrary function of q and r that satisfies all

requirements but one. We do this exercise only for *wlev* hence we do not consider requirement $e2$.

$$wlev.S.q.r = true, \text{ falsifies } e0$$
$$wlev.S.q.r = wlp.S.(q \vee r) \vee q \vee \neg[\neg q] \wedge \neg r, \text{ falsifies } e1$$
$$wlev.S.q.r = wlp.S.(q \vee r), \text{ falsifies } e3$$
$$wlev.S.q.r = q \vee [\neg q] \wedge wlp.S.(q \vee r), \text{ falsifies } e4$$
$$wlcv.S.q.r = q \vee wlp.S.r, \text{ falsifies } e5$$

The fact that the second one falsifies $e1$ may not be so easy to see. Choose $y := 4$ for S, *false* for q and $\{false, x = 3\}$ as a set of predicates over which *wlev* is not conjunctive.

These are the requirements that we use. They are verified in the next section for each construct in the language which, in fact, means that they are proven by structural induction. All results that we use or prove are based on these requirements. We can, therefore, proceed with some theorems.

The fact that ";" is associative is a theorem in the previous chapter. However, in the abstract way that we have defined *wlev* and *wev*, based on the syntax only, this associativity is not obvious. Furthermore, we can only prove that ";" is associative *with respect to wlev* and *wev*. (We might come up with another property and weakest precondition for which ";" is not associative. However, such a weakest precondition would not allow an implementation as described in the previous chapter.)

Theorem 65 *";" is associative with respect to wlev and wev.*

Proof.

$$\begin{array}{ll} & wlev.(S;(U;V)).q.r \\ = & \quad \{\text{definition}\} \\ & wlev.S.q.(wlev.(U;V).q.r) \\ = & \quad \{\text{definition}\} \\ & wlev.S.q.(wlev.U.q.(wlev.V.q.r)) \\ = & \quad \{\text{definition}\} \\ & wlev.(S;U).q.(wlev.V.q.r) \\ = & \quad \{\text{definition}\} \\ & wlev.((S;U);V).q.r \end{array}$$

The proof for *wev* is completely similar.

□

Sequential composition being associative is but one property, based on operational considerations, that we may guess. We prove two more. Since *skip* does nothing, i.e., it does not change the state, we have the following theorem.

Theorem 66 *With respect to wlev and wev we have for arbitrary S,*

$$S; skip = S = skip; S.$$

Proof. We only prove this for *wev*; the proof for *wlev* is similar.

$$
\begin{array}{ll}
 & wcv.(S; skip).q.r \\
= & \quad \{\text{definition ";"}\} \\
 & wev.S.q.(wev.skip.q.r) \\
= & \quad \{\text{definition } skip\} \\
 & wev.S.q.(q \vee r) \\
= & \quad \{e5'\} \\
 & wev.S.q.r \\
= & \quad \{e3'\} \\
 & q \vee wev.S.q.r \\
= & \quad \{\text{definition } skip\} \\
 & wev.skip.q.(wev.S.q.r) \\
= & \quad \{\text{definition ";"}\} \\
 & wev.(skip; S).q.r
\end{array}
$$

□

Theorem 67 *With respect to wlcv and wcv we have for arbitrary S,*

$$abort; S \;=\; abort.$$

Proof. Both *wlev* and *wev* are independent of their third arguments. With the definition of ";" the result now follows immediately.

□

Our next result relates $wlp, wp, wlev$ and wev. It comes in a lot of flavors.

Theorem 68 *For program S and predicates q, r and w, we have*

$$
\begin{array}{ll}
(i) & [wlev.S.q.r \wedge wlp.S.w \Rightarrow wlev.S.q.(r \wedge w)], \\
(ii) & [wlev.S.q.r \wedge wp.S.w \Rightarrow wev.S.q.(r \wedge w)], \\
(iii) & [wev.S.q.r \wedge wlp.S.w \Rightarrow wev.S.q.(r \wedge w)], \\
(iv) & [wev.S.q.r \wedge wp.S.w \Rightarrow wev.S.q.(r \wedge w)].
\end{array}
$$

Proof. We give the proof for only one of these. The others are similar. We choose (ii).

$$
\begin{array}{ll}
 & wlev.S.q.r \wedge wp.S.w \\
= & \quad \{e0'\} \\
 & wlev.S.q.r \wedge wev.S.false.w \\
\Rightarrow & \quad \{e4'\} \\
 & wlev.S.q.r \wedge wev.S.q.w \\
= & \quad \{e2, e1 \text{ and } e2 \text{ again}\} \\
 & wev.S.q.(r \wedge w)
\end{array}
$$

□

The implications in theorem 68 are for real; they cannot be replaced by equalities. A counterexample for (i) is the following program.

S: **if** $true \rightarrow x := 1; y := 2; z := 3$
[] $true \rightarrow y := 2; z := 4$
fi

Choose $q = (x = 1), r = (y = 2)$ and $w = (z = 4)$. We show that $wlev.S.q.r$ and $wlev.S.q.(r \wedge w)$ hold.

$wlev.S.(x = 1).(y = 2 \wedge z = 4)$
$=$ {definition if}
$wlev.(x := 1; y := 2; z := 3).(x = 1).(y = 2 \wedge z = 4) \ \wedge$
$wlev.(y := 2; z := 4).(x = 1).(y = 2 \wedge z = 4)$
$=$ {definition ";"and assignment, twice}
$wlev.(x := 1; y := 2).(x = 1).(x = 1) \ \wedge \ wlev.(y := 2).(x = 1).(x = 1 \vee y = 2)$
$=$ {definition ";" and assignment}
$wlev.(x := 1).(x = 1).(x = 1) \ \wedge \ true$
$=$ {definition assignment}
$true$

This also implies $[wlev.S.(x = 1).(y = 2)]$ since $wlev.S.q$ is monotonic. However, $wlp.S.(z = 4) = false$.

The next result can be viewed as an extension of the Main Repetition Theorem ([9]). For a repetition **do** $B \rightarrow S$ **od**, it reduces the proof of $wev.DO.q.r$ to proving something about S.

Theorem 69 *Consider the repetition* **do** $B \rightarrow S$ **od**. *Let t be a function from the state space to some set D and let $(C, <)$ be a well founded subset of D. I, q and r are predicates. If*

(i) $[I \wedge B \Rightarrow t \in C]$,
(ii) $\forall(x : x \in C : [I \wedge B \wedge t = x \Rightarrow wev.S.q.(I \wedge (B \vee q \vee r) \wedge t < x)]$,

then

$$[I \wedge (B \vee q \vee r) \Rightarrow wev.DO.q.r].$$

Proof. The proof looks a lot like the proof given in [9]. Abbreviate $wev.DO.q.r$ by z and the right-hand side of (6.2) by $G.Y$. Because of $e3'$, we only have to prove $[I \wedge (B \vee r) \ \Rightarrow \ z]$.

$[I \wedge (B \vee r) \ \Rightarrow \ z]$
$=$ {calculus}
$[I \wedge (B \vee r) \wedge (t \in C \vee t \notin C) \ \Rightarrow \ z]$
$=$ {distribute universal quantification over conjunction}
$[I \wedge (B \vee r) \wedge t \in C \ \Rightarrow \ z] \ \wedge \ [I \wedge (B \vee r) \wedge t \notin C \ \Rightarrow \ z]$

We prove that the second term is equivalent to *true*.

$$
\begin{array}{ll}
 & [I \wedge (B \vee r) \wedge t \notin C \;\Rightarrow\; z] \\
= & \quad \{\text{calculus}, B \vee r \equiv (B \vee (\neg B \wedge r))\} \\
 & [I \wedge B \wedge t \notin C \;\Rightarrow\; z] \;\wedge\; [I \wedge \neg B \wedge r \wedge t \notin C \;\Rightarrow\; z] \\
= & \quad \{(i) : [I \wedge B \wedge t \notin C \equiv \mathit{false}], z \equiv G.z\} \\
 & [I \wedge \neg B \wedge r \wedge t \notin C \;\Rightarrow\; G.z] \\
= & \quad \{\text{use } \neg B \wedge r \text{ in the definition of } G\} \\
 & \mathit{true}
\end{array}
$$

Hence,

$$[I \wedge (B \vee r) \;\Rightarrow\; z] \equiv [I \wedge (B \vee r) \wedge t \in C \;\Rightarrow\; z] \tag{6.3}$$

Our proof obligation boils down to

$$
\begin{array}{ll}
 & [I \wedge (B \vee r) \wedge t \in C \;\Rightarrow\; z] \\
= & \quad \{\text{one point rule}\} \\
 & [\forall (x : t = x : I \wedge (B \vee r) \wedge x \in C \;\Rightarrow\; z)] \\
= & \quad \{\text{trading}\} \\
 & [\forall (x : x \in C : I \wedge (B \vee r) \wedge t = x \;\Rightarrow\; z)] \\
= & \quad \{\text{interchange universal quantifications}\} \\
 & \forall (x : x \in C : [I \wedge (B \vee r) \wedge t = x \;\Rightarrow\; z]).
\end{array}
$$

We prove this by induction, i.e., we prove the above assuming

$$\forall (y : y \in C \wedge y < x : [I \wedge (B \vee r) \wedge t = y \;\Rightarrow\; z]).$$

Here we need to use (ii) but in a different form than it is given. First we notice that, because of $e5'$ (twice), (ii) can be rephrased as

$$(ii) \quad \forall (x : x \in C : [I \wedge B \wedge t = x \;\Rightarrow\; \mathit{wev.S.q.}(I \wedge (B \vee r) \wedge t < x)]).$$

We need to get rid of $\mathit{wev.S.q.}(I \wedge (B \vee r) \wedge t < x)$ and replace it by G applied to some argument. For $x \in C$ we have

$$
\begin{array}{ll}
 & I \wedge B \wedge t = x \;\Rightarrow\; \mathit{wev.S.q.}(I \wedge (B \vee r) \wedge t < x) \\
= & \quad \{\text{calculus}\} \\
 & I \wedge B \wedge t = x \;\Rightarrow\; \mathit{wev.S.q.}(I \wedge (B \vee r) \wedge t < x) \wedge B \\
= & \quad \{\text{calculus}\} \\
 & I \wedge (B \vee q \vee r) \wedge (B \vee \neg(q \vee r)) \wedge t = x \;\Rightarrow\; \mathit{wev.S.q.}(I \wedge (B \vee r) \wedge t < x) \wedge B \\
= & \quad \{\text{calculus}\} \\
 & I \wedge (B \vee q \vee r) \wedge t = x \;\Rightarrow\; (\neg B \wedge (q \vee r)) \vee (\mathit{wev.S.q.}(I \wedge (B \vee r) \wedge t < x) \wedge B) \\
= & \quad \{\text{definition } G\} \\
 & I \wedge (B \vee q \vee r) \wedge t = x \;\Rightarrow\; G.(I \wedge (B \vee r) \wedge t < x) \\
\Rightarrow & \quad \{\text{strengthening the antecedent}\} \\
 & I \wedge (B \vee r) \wedge t = x \;\Rightarrow\; G.(I \wedge (B \vee r) \wedge t < x).
\end{array}
$$

Hence,

$$\forall(x : x \in C : [I \wedge (B \vee r) \wedge t = x \;\Rightarrow\; G.(I \wedge (B \vee r) \wedge t < x)]). \tag{6.4}$$

Now we are ready to finish the proof. We start with our induction hypothesis. For $x \in C$ we have

$$\begin{array}{ll}
 & \forall(y : Y \in C \wedge y < x : [I \wedge (B \vee r) \wedge t = y \;\Rightarrow\; z]) \\
= & \quad \{\text{interchange of quantifications}\} \\
 & [\forall(y : y \in C \wedge y < x : I \wedge (B \vee r) \wedge t = y \;\Rightarrow\; z)] \\
= & \quad \{\text{trading}\} \\
 & [\forall(y : t = y : I \wedge (B \vee r) \wedge y \in C \wedge y < x \;\Rightarrow\; z)] \\
= & \quad \{\text{one point rule}\} \\
 & [I \wedge (B \vee r) \wedge t \in C \wedge t < x \;\Rightarrow\; z] \\
= & \quad \{(6.3)\} \\
 & [I \wedge (B \vee r) \wedge t < x \;\Rightarrow\; z] \\
\Rightarrow & \quad \{G \text{ is monotonic}\} \\
 & [G.(I \wedge (B \vee r) \wedge t < x) \;\Rightarrow\; G.z] \\
\Rightarrow & \quad \{(6.4) \text{ and } x \in C\} \\
 & [I \wedge (B \vee r) \wedge t = x \;\Rightarrow\; G.z] \\
= & \quad \{z \equiv G.z\} \\
 & [I \wedge (B \vee r) \wedge t = x \;\Rightarrow\; z].
\end{array}$$

□

Notice that this is a generalization of the Main Repetition Theorem. By choosing $q = \mathit{false}$ and $r = (I \wedge \neg B)$ we obtain exactly that theorem.

Next we give another example, an application of theorem 69. Consider the program

$$S : x := 10;\ \mathbf{do}\ x \neq 0 \rightarrow x := x - 1\ \mathbf{od}.$$

We prove $[\mathit{wev}.S.(x = 3).\mathit{false}]$, i.e., at some point during execution of S, x will have the value 3. We call the repetition U and we use theorem 69 with the following instantiations. $I = 3 \leq x \leq 10, t = x, C = D = \mathbb{N}$. We are going to prove $[I \Rightarrow \mathit{wev}.U.(x = 3).\mathit{false}]$. Then the result follows from

$$\begin{array}{ll}
 & \mathit{wev}.S.(x = 3).\mathit{false} \\
= & \quad \{\text{definition ";"}\} \\
 & \mathit{wev}.(x := 10).(x = 3).(\mathit{wev}.U.(x = 3).\mathit{false}) \\
\Leftarrow & \quad \{\mathit{wev}.S.q \text{ is a monotonic function}\} \\
 & \mathit{wev}.(x := 10).(x = 3).I \\
= & \quad \{\text{definition assignment}\} \\
 & \mathit{true}.
\end{array}$$

We check the proof obligations, (i) and (ii).

$$
\begin{array}{lll}
(i): & & [3 \leq x \leq 10 \wedge x \neq 0 \;\Rightarrow\; x \in I\!N] \\
& = & \quad \{\text{calculus}\} \\
& & \quad \mathit{true} \\
(ii): & & \mathit{wev}.(x := x-1).(x = 3).(I \wedge x < T) \\
& = & \quad \{\text{definition assignment}\} \\
& & \quad x = 3 \vee x - 1 = 3 \vee (3 \leq x - 1 \leq 10 \wedge x - 1 < T) \\
& = & \quad \{\text{arithmetic}\} \\
& & \quad x = 3 \vee x = 4 \vee (4 \leq x \leq 11 \wedge x - 1 < T) \\
& \Leftarrow & \quad \{\text{calculus, arithmetic}\} \\
& & \quad 3 \leq x \leq 10 \wedge x = T
\end{array}
$$

for all values of T.
Remark. Notice that the weaker "invariant" $3 \leq x$ also does the job.

Finally, we have another look at nondeterminism. We recall from the previous chapter that S is deterministic with respect to *wlev* if

$$[\neg \mathit{wlev}.S.q.r \equiv \mathit{wto}.S.q.\mathit{false}.\mathit{false} \wedge \mathit{wp}.S.\neg r]$$

and with respect to *wev* if

$$[\neg \mathit{wev}.S.q.r \equiv \mathit{wto}.S.q.\mathit{false}.\mathit{false} \wedge \mathit{wlp}.S.\neg r].$$

We consider again the program from the previous chapter.

S: **if** $\mathit{true} \rightarrow x := 3; y := 4$
[] $\mathit{true} \rightarrow y := 4; x := 3$
fi

Using *wlp* and *wp* only, both choices are indistinguishable and, hence, this program is deterministic with respect to *wlp* and *wp*. We show that this program is nondeterministic with respect to both *wlev* and *wev* by choosing $q : x = 3 \wedge y = 5$ and $r : \mathit{false}$. Since we have only terminating statements here, *wlev* and *wev* coincide. Because we only look at final condition *false* for terminating statements, both the *wlp* and the *wp* in the above formula vanish.

$$
\begin{array}{ll}
& \mathit{wev}.S.(x = 3 \wedge y = 5).\mathit{false} \\
= & \quad \{\text{definition } \mathit{wev} \text{ for } \mathit{IF}\} \\
& \mathit{wev}.(x := 3; y := 4).(x = 3 \wedge y = 5).\mathit{false} \wedge \mathit{wev}.(y := 4; x := 3).(x = 3 \wedge y = 5).\mathit{false} \\
= & \quad \{\text{definition } \mathit{wev} \text{ for ";"}\} \\
& \mathit{wev}.(x := 3).(x = 3 \wedge y = 5).(\mathit{wev}.(y := 4).(x = 3 \wedge y = 5).\mathit{false}) \wedge \\
& \mathit{wev}.(y := 4).(x = 3 \wedge y = 5).(\mathit{wev}.(x := 3).(x = 3 \wedge y = 5).\mathit{false}) \\
= & \quad \{\text{definition } \mathit{wev} \text{ for assignment}\}
\end{array}
$$

$$
\begin{array}{ll}
 & wev.(x := 3).(x = 3 \wedge y = 5).(x = 3 \wedge y = 5) \wedge wev.(y := 4).(x = 3 \wedge y = 5).(y = 5) \\
= & \quad \{\text{definition } wev \text{ for assignment}\} \\
 & x = 3 \wedge y = 5
\end{array}
$$

$$
\begin{array}{ll}
 & wto.S.(x = 3 \wedge y = 5).false.false \\
= & \quad \{\text{definition } wto \text{ for } IF\} \\
 & wto.(x := 3; y := 4).(x = 3 \wedge y = 5).false.false \wedge \\
 & wto.(y := 4; x := 3).(x = 3 \wedge y = 5).false.false \\
= & \quad \{\text{definition } wto \text{ for ";"}\} \\
 & wto.(x := 3).(x = 3 \wedge y = 5).false.(wev.(y := 4).false.false) \wedge \\
 & \qquad wlp.(x := 3).(wto.(y := 4).(x = 3 \wedge y = 5).false.false) \wedge \\
 & wto.(y := 4).(x = 3 \wedge y = 5).false.(wev.(x := 3).false.false) \wedge \\
 & \qquad wlp.(y := 4).(wto.(x := 3).(x = 3 \wedge y = 5).false.false) \\
= & \quad \{\text{definition } wto \text{ and } wev \text{ for assignment}\} \\
 & wto.(x := 3).(x = 3 \wedge y = 5).false.false \wedge wlp.(x := 3).(x \neq 3 \vee y \neq 5) \\
 & wto.(y := 4).(x = 3 \wedge y = 5).false.false \wedge wlp.(y := 4).(y \neq 5) \\
= & \quad \{\text{definition } wto \text{ for assignment}\} \\
 & y \neq 5 \wedge (x \neq 3 \vee y \neq 5) \\
= & \quad \{\text{calculus}\} \\
 & y \neq 5
\end{array}
$$

Since $\neg(x = 3 \wedge y = 5)$ is not the same as $y \neq 5$, S is not deterministic. If we take $S; abort$, where S is as above, we have an example of a program that is deterministic with respect to $wlev$ but nondeterministic with respect to wev.

6.3 Proofs of the requirements

In the introduction of this chapter we argued that some requirements must be satisfied if a pair of functions is to be seen as the $wlev$ and the wev of a statement. In this section, we prove that the requirements, introduced in the previous section, are satisfied by the statements in our language. The requirements that we must verify are the following.

$e0$: $[wlev.S.false.r \equiv wlp.S.r]$

$e0'$: $[wev.S.false.r \equiv wp.S.r]$

$e1$: $wlev.S.q$ is universally conjunctive

$e2$: $[wev.S.q.r \equiv wev.S.q.true \ \wedge wlev.S.q.r]$

$e3'$: $[q \Rightarrow wev.S.q.r]$

$e4$: $[q \Rightarrow q'] \Rightarrow [wlev.S.q.r \Rightarrow wlev.S.q'.r]$

$e4'$: $[q \Rightarrow q'] \Rightarrow [wev.S.q.r \Rightarrow wev.S.q'.r]$

$e5:\ [wlev.S.q.r \equiv wlev.S.q.(q \vee r)]$

Because of $e0$ and $e0'$ we give here the definitions of wlp and wp.

$$
\begin{array}{ll}
wlp.skip.r = r & wp.skip.r = r \\
wlp.abort.r = true & wp.abort.r = false \\
wlp.(y := e).r = r_e^y & wp.(y := e).r = r_e^y \\
wlp.(S;U).r = wlp.S.(wlp.U.r) & wp.(S;U).r = wp.S.(wp.U.r)
\end{array}
$$

$$
\begin{array}{l}
wlp.(\textbf{if}\ [](i :: B_i \rightarrow S_i)\ \textbf{fi}).r = \forall(i : B_i : wlp.S_i.r) \\
wp.(\textbf{if}\ [](i :: B_i \rightarrow S_i)\ \textbf{fi}).r = \exists(i :: B_i) \wedge \forall(i : B_i : wp.S_i.r)
\end{array}
$$

The two equations for DO are

$$[Y \equiv (\neg B \vee wlp.S.Y)\ \wedge\ (B \vee r)] \tag{6.5}$$

$$[Y \equiv (\neg B \vee wp.S.Y)\ \wedge\ (B \vee r)] \tag{6.6}$$

$wlp.S.r$ is the weakest solution of (6.5) and $wp.S.r$ the stongest solution of (6.6).

In the following subsections, V is a set of predicates. In the proofs for composite structures, we may use the requirements for the components as induction hypothesis. Moreover, we may use the results of the previous section for these components.

6.3.1 *skip*

$e0:$ follows immediately from the definitions

$e0':$ similar

$e1:$ $[\forall(v : v \in V : wlev.skip.q.v) \equiv wlev.skip.q.\forall(v : v \in V : v)]$

$=$ {definition}

$[\forall(v : v \in V : q \vee v) \equiv q \vee \forall(v : v \in V : v)]$

$=$ {disjunction with a constant distributes over universal quantification}

$true$

$e2:$ $[wev.skip.q.r \equiv wev.skip.q.true \wedge wlev.skip.q.r]$

$=$ {definition}

$[q \vee r \equiv true \wedge (q \vee r)]$

$e3':$ follows from $[q \Rightarrow q \vee r]$

$e4:$ follows from $[q \Rightarrow q'] \Rightarrow [(q \vee r) \Rightarrow (q' \vee r)]$

$e4':$ similar

$e5:$ follows from $q \vee r\ \equiv\ q \vee q \vee r$

6.3.2 *abort*

$e0$: $[wlev.abort.false.r \equiv wlp.abort.r]$

$=$ {definition}

$[true \equiv true]$

$e0'$: $[wev.abort.false.r \equiv wp.abort.r]$

$=$ {definition}

$[false \equiv false]$

$e1$: $[\forall(v : v \in V : wlev.abort.q.v) \equiv wlev.abort.q.(\forall(v : v \in V : v))]$

$=$ {definition}

$[\forall(v : v \in V : true) \equiv true]$

$=$ {calculus}

$true$

$e2$: $[wev.abort.q.r \equiv wev.abort.q.true \wedge wlev.abort.q.r]$

$=$ {definition}

$[q \equiv q \wedge true]$

$e3'$: follows from $[q \Rightarrow q]$

$e4$: follows from $[true \Rightarrow true]$

$e4'$: follows from $[q \Rightarrow q'] \Rightarrow [q \Rightarrow q']$

$e5$: both functions do not depend on their third arguments

6.3.3 y:=e

$e0$: $[wev.(y := e).false.r \equiv wlp.(y := e).r]$

$=$ {definition}

$[r_e^y \equiv r_e^y]$

$e0'$: same reasoning

$e1$: $[\forall(v : v \in V : wlev.(y := e).q.v) \equiv wlev.(y := e).q.(\forall(v : v \in V : v))]$

$=$ {definition}

$[\forall(v : v \in V : q \vee q_e^y \vee v_e^y) \equiv q \vee q_e^y \vee \forall(v : v \in V : v)_e^y]$

$=$ {substitution and disjunction distribute over universal quantification}

$true$

$e2$: $[wev.(y := e).q.r \equiv wev.(y := e).q.true \wedge wlev.(y := e).q.r]$

$=$ {definition}

$[q \vee q_e^y \vee r_e^y \equiv true \wedge (q \vee q_e^y \vee r_e^y)]$

$e3'$: follows from $[q \Rightarrow q \vee q_e^y \vee r_e^y]$

$e4$: substitution is monotonic

$e4'$: same argument

$e5$: follows from $[q \vee q_c^y \vee r_c^y \equiv q \vee q_c^y \vee (q_c^y \vee r_c^y)]$

6.3.4 $S;U$

$e0$: $wlev.(S;U).false.r$

$=$ {definition}

$wlev.S.false.(wlev.U.false.r)$

$=$ {$e0$ for U}

$wlev.S.false.(wlp.U.r)$

$=$ {$e0$ for S}

$wlp.S.(wlp.U.r)$

$=$ {definition}

$wlp.(S;U).r$

$e0'$: similar reasoning

$e1$: $\forall(v : v \in V : wlev.(S;U).q.v)$

$=$ {definition}

$\forall(v : v \in V : wlev.S.q.(wlev.U.q.v))$

$=$ {$e1$ for S}

$wlev.S.q.(\forall(v : v \in V : wlev.U.q.v))$

$=$ {$e1$ for U}

$wlev.S.q.(wlev.U.q.(\forall(v : v \in V : v)))$

$=$ {definition}

$wlev.(S;U).q.(\forall(v : v \in V : v))$

$e2$: $wev.(S;U).q.r$

$=$ {definition}

$wcv.S.q.(wcv.U.q.r)$

$=$ {$e2$ for U}

$wev.S.q.(wev.U.q.true \wedge wlev.U.q.r)$

$=$ {$e1'$ for S}

$wev.S.q.(wev.U.q.true) \wedge wev.S.q.(wlev.U.q.r)$

$=$ {$e2$ for S}

$wev.S.q.(wev.U.q.true) \wedge wev.S.q.true \wedge wlev.S.q.(wlev.U.q.r)$

$=$ {$e1'$ for S, definition}

$wev.(S;U).q.true \wedge wlev.(S;U).q.r$

$e3'$: $wev.(S;U).q.r$

$=$ {definition}

$wev.S.q.(wev.U.q.r)$

$\Leftarrow$ {$e3'$ for S}

q

$e4$: $[wlev.(S;U).q.r \Rightarrow wlev.(S;U).q'.r]$

$=$ {definition}

$[wlev.S.q.(wlev.U.q.r) \Rightarrow wlev.S.q'.(wlev.U.q'.r)]$

$\Leftarrow$ {$e1$ for S: conjunctivity implies monotonicity, $e4$ for U}

$[wlev.S.q.(wlev.U.q'.r) \Rightarrow wlev.S.q'.(wlev.U.q'.r)] \wedge [q \Rightarrow q']$

$=$ {$e4$ for S}

$[q \Rightarrow q']$

$e4'$: similar reasoning: $e1'$ also implies monotonicity

$e5$: $wev.(S;U).q.r$

$=$ {definition}

$wev.S.q.(wev.U.q.r)$

$=$ {$e5$ for U}

$wev.S.q.(wev.U.q.(q \vee r))$

$=$ {definition}

$wev.(S;U).q.(q \vee r)$

6.3.5 if $[](i :: B_i \rightarrow S_i)$ fi

We abbreviate **if** $[](i :: B_i \rightarrow S_i)$ **fi** by *IF*.

$e0$: $[wlev.IF.false.r \equiv wlp.IF.r]$

$=$ {definition, $e0$ for the S_i}

$[\forall(i : B_i : wlp.S_i.r) \equiv wlp.IF.r]$

$=$ {definition}

true

$e0'$: $[wev.IF.false.r \equiv wp.IF.r]$

$=$ {definition, $e0'$ for the S_i}

$[\forall(i : B_i : wp.S_i.r) \wedge \exists(i :: B_i) \equiv wp.IF.r]$

$=$ {definition}

true

$e1$: $\forall(v : v \in V : wlev.IF.q.v)$

$=$ {definition}

$\forall(v : v \in V : \forall(i : B_i : wlev.S_i.q.v))$

$=$ {calculus}

$\forall(i : B_i : \forall(v : v \in V : wlev.S_i.q.v))$

$=$ $\{e1$ for the $S_i\}$
$\forall(i : B_i : wlev.S_i.q.(\forall(v : v \in V : v)))$
$=$ {definition}
$wlev.IF.q.(\forall(v : v \in V : v))$

$e2:$ $wev.IF.q.r$
$=$ {definition}
$\forall(i : B_i : wev.S_i.q.r) \wedge (q \vee \exists(i :: B_i))$
$=$ $\{e2$ for the $S_i\}$
$\forall(i : B_i : wev.S_i.q.true \wedge wlev.S_i.q.r) \wedge (q \vee \exists(i :: B_i))$
$=$ {calculus}
$\forall(i : B_i : wlev.S_i.q.r) \wedge \forall(i : B_i : wev.S_i.q.true) \wedge (q \vee \exists(i :: B_i))$
$=$ {definition}
$wlev.IF.q.r \wedge wev.IF.q.true$

$e3':$ $[q \Rightarrow wev.IF.q.r]$
$=$ {definition}
$[q \Rightarrow \forall(i : B_i : wev.S_i.q.r) \wedge (q \vee \exists(i :: B_i))]$
$=$ $\{e3'$ for the S_i, calculus$\}$
$true$

$e4:$ similar to (even simpler than) $e4'$

$e4':$ $[wev.IF.q.r \Rightarrow wev.IF.q'.r]$
$=$ {definition}
$[\forall(i : B_i : wev.S_i.q.r) \wedge (q \vee \exists(i :: B_i)) \Rightarrow$
$\forall(i : B_i : wev.S_i.q'.r) \wedge (q' \vee \exists(i :: B_i))]$
$\Leftarrow$ {calculus}
$[\forall(i : B_i : wev.S_i.q.r) \Rightarrow \forall(i : B_i : wev.S_i.q'.r)] \wedge [q \Rightarrow q']$
$=$ $\{e4'$ for the $S_i\}$
$[q \Rightarrow q']$

$e5:$ immediate from $e5$ for the S_i

6.3.6 do $B \rightarrow S$ od

We abbreviate **do** $B \rightarrow S$ **od** by DO. The proofs for DO will be different from the ones given above since $wlev$ and wev for DO are defined as solutions of recursive equations. In order to prove $e0$ and $e0'$, we show that, for $q = false$, (6.1) and (6.2) reduce to (6.5) and (6.6) respectively. Since both wlp and wev are weakest solutions and both wev and wp are strongest solutions, the result then follows.

(6.1) for $q = false$
$=$

$$
\begin{array}{ll}
 & [Y \equiv (\neg B \vee wlev.S.false.Y) \ \wedge \ (B \vee r)] \\
- & \quad \{e0 \text{ for } S\} \\
 & [Y \equiv (\neg B \vee wlp.S.Y) \ \wedge \ (B \vee r)] \\
= & \\
 & (6.5)
\end{array}
$$

The reasoning for $e0'$ is similar. To show the validity of $e1$, we apply theorem 31. To this end we have to show that $F.q.r.Y$ is $\{1,2\}$-universally conjunctive. (Remember that F and G were defined to be the right-hand sides of (6.1) and (6.2) respectively.) Let V be a $\{1,2\}$-projection (in P^3), i.e., all elements of V have the same first component. Hence, we may substitute $\forall(v : v \in V : v.0)$ by $v.0$ for an arbitrary $v \in V$. We must show,

$$[\forall(v : v \in V : F.v) = F.(\forall(v : v \in V : v))].$$

Notice that boolean operators on vectors of predicates are applied componentwise. This follows from theorem 25.

$$
\begin{array}{ll}
 & \forall(v : v \in V : F.v) \\
= & \quad \{\text{definition } F\} \\
 & \forall(v : v \in V : (\neg B \vee wlev.S.(v.0).(v.2)) \wedge (B \vee v.0 \vee v.1)) \\
= & \quad \{\text{distribute universal quantification}\} \\
 & \forall(v : v \in V : \neg B \vee wlev.S.(v.0).(v.2)) \ \wedge \ \forall(v : v \in V : B \vee v.0 \vee v.1) \\
= & \quad \{e1 \text{ for } S, \forall(v \in V : v.0 = \forall(v : v \in V : v.0)), \text{calculus}\} \\
 & (\neg B \vee wlev.S.(\forall(v : v \in V : v.0)).(\forall(v : v \in V : v.2))) \ \wedge \\
 & \qquad (B \vee \forall(v : v \in V : v.0) \vee \forall(v : v \in V : v.1)) \\
= & \quad \{\text{definition and } \forall(v : v \in V : v.i) = \forall(v : v \in V : v).i\} \\
 & F.(\forall(v : v \in V : v))
\end{array}
$$

We proceed with $e2$. It seems that we have a problem here because we have to relate solutions of different equations. We use $e2$ for S to bring (6.1) and (6.2) a bit closer together.

$$[Y \equiv (\neg B \vee wlev.S.q.Y) \ \wedge \ (\neg B \vee wev.S.q.true) \ \wedge \ (B \vee q \vee r)]$$

Notice that the only difference with (6.1) is the extra term $\neg B \vee wev.S.q.true$. We account for that term by introducing an extra argument. We define the function H.

$$H.q.r.z.Y = (\neg B \vee wlev.S.q.Y) \ \wedge \ (B \vee q \vee r) \ \wedge \ z$$

$H.q.r.z$ is a monotonic function hence theorem 23 applies. Let g denote the strongest and h the weakest solution of $[Y \equiv H.q.r.z.Y]$. (They are the smallest and the greatest solutions respectively in the order defined by $\Rightarrow$, hence, this is consistent with the nomenclature in chapter four.) Now we have,

$$
\begin{array}{l}
wlev.DO.q.r = h.q.r.true, \\
wev.DO.q.r = g.q.r.(\neg B \vee wev.S.q.true).
\end{array}
$$

We want to apply theorem 36. Therefore, we have to prove that H is $\{1,2,3\}$-finitely conjunctive which boils down to proving,

$$H.q.(r \wedge r').(z \wedge z').(Y \wedge Y') \;\equiv\; H.q.r.z.Y \wedge H.q.r'.z'.Y'.$$

This follows immediately from the definition of H and the (universal) conjunctivity of $wlev.S.q$. The application of theorem 36 yields

$$[h.q.r.true \wedge g.q.true.(\neg B \vee wev.S.q.true) \equiv g.q.r.(\neg B \vee wev.S.q.true)].$$

This proves $e2$. For $e3'$ observe

$$\begin{array}{ll} & [q \Rightarrow wev.DO.q.r] \\ = & \quad \{wev.DO.q.r \text{ solves } (6.2)\} \\ & [q \Rightarrow (\neg B \vee wev.S.q.(wev.DO.q.r)) \;\wedge\; (B \vee q \vee r)] \\ = & \quad \{\text{use } e3' \text{ for } S\} \\ & true \end{array}$$

Both F and G are $\{0,2\}$-monotonic, because of $e4$ and $e4'$ for S. By theorem 30, $wlev.DO.q.r$ and $wev.DO.q.r$ are $\{0\}$ monotonic. Finally, $e5$: observe that (6.1) does not change by substituting $q \vee r$ for r.

Chapter 7

The property leads-to.$p.q$

7.1 Introduction

In this chapter we explore the functions *wlto* and *wto* that define the property *leads-to.p.q*. We proceed in the same way as in the previous chapter. In the next section we introduce some requirements that distinguish functions that can be the *wlto* and *wto* of a statement from those that cannot. From these requirements we derive some theorems. In the third section the proofs are given that our programming language satisfies the requirements. First we recall the definitions of the two functions.

$$wlto.skip.p.q.r = \neg p \vee q \vee r \qquad wto.skip.p.q.r = \neg p \vee q \vee r$$
$$wlto.abort.p.q.r = true \qquad wto.abort.p.q.r = p \Rightarrow q$$

$$wlto.(y := e).p.q.r = (\neg p \vee q \vee q_e^y \vee r_e^y) \wedge (\neg p_e^y \vee q_e^y \vee r_e^y)$$
$$wto.(y := e).p.q.r = (\neg p \vee q \vee q_e^y \vee r_e^y) \wedge (\neg p_e^y \vee q_e^y \vee r_e^y)$$
$$wlto.(S;U).p.q.r = wlto.S.p.q.(wlev.U.q.r) \wedge wlp.S.(wlto.U.p.q.r)$$
$$wto.(S;U).p.q.r = wto.S.p.q.(wev.U.q.r) \wedge wlp.S.(wto.U.p.q.r)$$
$$wlto.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).p.q.r = \forall(i : B_i : wlto.S_i.p.q.r)$$
$$wto.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).p.q.r = ((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q.r)$$

For **do** $B \rightarrow S$ **od**, *wlto* and *wto* are the weakest solutions of two equations in unknown predicate Y.

$$[Y \equiv B \wedge wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.Y \ \vee \ \neg B \wedge (\neg p \vee q \vee r)]$$
$$[Y \equiv B \wedge wto.S.p.q.(wev.DO.q.r) \wedge wlp.S.Y \ \vee \ \neg B \wedge (\neg p \vee q \vee r)]$$

We rewrite these equations as follows.

$$[Y \equiv (\neg B \vee wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.Y) \ \wedge \ (B \vee \neg p \vee q \vee r)] \tag{7.1}$$
$$[Y \equiv (\neg B \vee wto.S.p.q.(wev.DO.q.r) \wedge wlp.S.Y) \ \wedge \ (B \vee \neg p \vee q \vee r)] \tag{7.2}$$

wlto.DO.p.q.r is the weakest solution of (7.1) and *wto.DO.p.q.r* is the weakest solution of (7.2). We use the names F and G again for the right-hand sides of (7.1) and (7.2) respectively.

7.2 Requirements for *wlto* and *wto*

We used the functions *leads-to.p.q*, *lt.r* and *t.r* to define *wlto* and *wto*. Using these functions, the set of computations is partitioned into four classes.

"p leads-to q":	all computations such that every state in which p holds is eventually followed by one in which q holds.
"p without q and eternal":	all infinite computations in which there exists a state in which p holds which is not followed by one in which q holds.
"p without q and finally r":	all computations in which there exists a state in which p holds which is not followed by one in which q holds and which terminate in r.
"p without q and finally $\neg r$":	all computations in which there exists a state in which p holds which is not followed by one in which q holds and which terminate in $\neg r$.

In "followed by" we include coincidence. Using this classification, we can restate the meaning of *wlto* and *wto*. *wlto.S.p.q.r* holds exactly in those initial states for which no computation of S belongs to the class "p without q and finally $\neg r$". *wto.S.p.q.r* holds exactly in those initial states for which no computation of S belongs to the class "p without q and finally $\neg r$" or to the class "p without q and eternal". From this classification we can see that the function *wto.S.p.q.false* is the weakest precondition for S such that S has the property *leads-to.p.q*.

Again, we employ this classification to obtain our requirements. We choose constants for p, q and r and try to relate *wlto* and *wto* to functions we already know. By substituting *false* for q and *true* for p, the class "p leads-to q" becomes empty and the partitioning of the class "p without q" becomes the same as the one underlying the definitions of *wlp* and *wp*. For *wlto* we have an even stronger requirement. $wlto.S.p.q.(\neg p \vee q \vee r)$ holds exactly in those initial states such that no execution of S belongs to the class "p without q and finally $p \wedge \neg q \wedge \neg r$". If a computation terminates in a state satisfying $p \wedge \neg q \wedge \neg r$, it is in the class "$p$ without q", since for the final state p holds but q does not. Hence, $wlto.S.p.q.(\neg p \vee q \vee r)$ holds exactly in those initial states such that no execution of S belongs to the class "finally $p \wedge \neg q \wedge \neg r$".

$l0$: $[wlto.S.p.q.(\neg p \vee q \vee r) \equiv wlp.S.(\neg p \vee q \vee r)]$

$l0'$: $[wto.S.true.false.r = wp.S.r]$

This relation suggest the next requirement.

$l1$: *wlto.S.p.q* is universally conjunctive

This has as a special case, $[wlto.S.p.q.true]$. We require a similar relation between *wlto* and *wto* as we have for *wlev* and *wev*.

$l2$: $[wto.S.p.q.r \equiv wto.S.p.q.true \wedge wlto.S.p.q.r]$

Together with $l1$ this gives us $l1'$ as a corollary.

$l1'$: *wto.S.p.q* is positively conjunctive

Requirement $e3$ concerns the state in which a program is started. The corresponding requirement for *leads-to* is that if a state satisfying p is reached then this state is one that coincides with or is followed by a state for which p holds. In other words, every computation is in "p leads-to p".

$$l3' : \ [wto.S.p.p.r]$$

From $l3'$ and $l2$ we derive

$$l3 : \ [wlto.S.p.p.r].$$

Similar to the requirement for *wlev* and *wev* we require monotonicity in the third argument.

$$l4 : \ [q \Rightarrow q'] \Rightarrow [wlto.S.p.q.r \Rightarrow wlto.S.p.q'.r]$$
$$l4' : \ [q \Rightarrow q'] \Rightarrow [wto.S.p.q.r \Rightarrow wto.S.p.q'.r]$$

Combining $l3(l3')$ and $l4(l4')$ yields

$$[p \Rightarrow q] \Rightarrow [wlto.S.p.q.r],$$
$$[p \Rightarrow q] \Rightarrow [wto.S.p.q.r].$$

$e5$ concerns the final state of the program. We have a similar requirement for *wlto* and *wto* since a computation in the class "p without q", does not terminate in a state satisfying q because, in that case, the computation would be in the class "p leads-to q" for arbitrary p.

$$l5 : \ [wlto.S.p.q.r \equiv wlto.S.p.q.(q \vee r)]$$

Together with $l2$ this gives

$$l5' : \ [wto.S.p.q.r \equiv wto.S.p.q.(q \vee r)].$$

Next we turn to the second argument. Consider a computation in two classes: "$p0$ leads-to q" and "$p1$ leads-to q". This means that if either one of $p0$ and $p1$ holds, q becomes *true* later. The other implication holds as well. Let V be a set of predicates.

$$l6 : \ [\forall(v : v \in V : wlto.S.v.q.r) \equiv wlto.S.(\exists(v : v \in V : v)).q.r]$$
$$l6' : \ [\forall(v : v \in V : wto.S.v.q.r) \equiv wto.S.(\exists(v : v \in V : v)).q.r]$$

The next two requirements capture the most important characteristics of *wlto* and *wto*. *wlto* and *wto* are used to define a *progress* property, viz. *leads-to.p.q*. The progress is captured in two requirements, both expressing some sort of transitivity. The first one relates *wlev*, *wev* and *wlto*, *wto*. A computation in the class "p leads-to q" that is also in the class "ever p" is in the class "ever q".

$$l7 : \ [wlto.S.p.q.r \wedge wlev.S.p.r \Rightarrow wlev.S.q.r]$$
$$l7' : \ [wto.S.p.q.r \wedge wev.S.p.r \Rightarrow wev.S.q.r]$$

The last requirement is based on the observation that a computation in the class "p leads-to q" that is also in the class "q leads-to w" is in the class "p leads-to w".

$$l8 : \ [wlto.S.p.q.r \wedge wlto.S.q.w.r \Rightarrow wlto.S.p.w.r]$$
$$l8' : \ [wto.S.p.q.r \wedge wto.S.q.w.r \Rightarrow wto.S.p.w.r]$$

These are the requirements that we are going to work with. The number of requirements more or less reflects the fact that *wlto* and *wto* are functions with many arguments. The proof that our programming language satisfies these requirements is given in the next section. In the remaining part of this section, we prove some facts about *wlto* and *wto* using the above requirements.

Theorem 70

$$(i) \quad [p \Rightarrow p'] \Rightarrow [wlto.S.p'.q.r \Rightarrow wlto.S.p.q.r]$$
$$(ii) \quad [p \Rightarrow p'] \Rightarrow [wto.S.p'.q.r \Rightarrow wto.S.p.q.r]$$

Proof. (Only (i).)

$$\begin{array}{ll} & wlto.S.p.q.r \wedge wlto.S.p'.q.r \\ = & \quad \{l6\} \\ & wlto.S.(p \vee p').q.r \\ = & \quad \{[p \Rightarrow p']\} \\ & wlto.S.p'.q.r \end{array}$$

□

Next, we follow the same steps as in the previous chapter and prove the counterparts of theorems 65 through 69.

Theorem 71 *";" is associative with respect to wlto and wto.*

Proof. (Only for *wto*.)

$$\begin{array}{ll} & wto.(S;(U;V)).p.q.r \\ = & \quad \{\text{definition}\} \\ & wto.S.p.q.(wev.(U;V).q.r) \wedge wlp.S.(wto.(U;V).p.q.r) \\ = & \quad \{\text{definition}\} \\ & wto.S.p.q.(wev.U.q.(wev.V.q.r)) \wedge wlp.S.(wto.U.p.q.(wev.V.q.r) \wedge wlp.U.(wto.V.p.q.r)) \\ = & \quad \{\text{conjunctivity of } wlp\} \\ & wto.S.p.q.(wev.U.q.(wev.V.q.r)) \wedge wlp.S.(wto.U.p.q.(wev.V.q.r)) \wedge \\ & wlp.S.(wlp.U.(wto.V.p.q.r)) \\ = & \quad \{\text{definition}\} \\ & wto.(S;U).p.q.(wev.V.q.r) \wedge wlp.(S;U).(wto.V.p.q.r) \\ = & \quad \{\text{definition}\} \\ & wto.((S;U);V).p.q.r \end{array}$$

□

Before we prove that *skip* is the neutral element of sequential composition, we need a little lemma.

Lemma 72

(i) $[wlto.S.p.q.r \Rightarrow \neg p \vee wlev.S.q.r]$
(ii) $[wto.S.p.q.r \Rightarrow \neg p \vee wev.S.q.r]$

Proof. (Only (ii).)

$[wto.S.p.q.r \Rightarrow \neg p \vee wev.S.q.r]$
$=$ {calculus}
$[wto.S.p.q.r \wedge p \Rightarrow wev.S.q.r]$
$\Leftarrow$ {$e3'$}
$[wto.S.p.q.r \wedge wev.S.p.r \Rightarrow wev.S.q.r]$
$=$ {$l7'$}
$true$

□

Theorem 73 *With respect to wlto and wto we have*

$$S; skip \;=\; S \;=\; skip; S.$$

Proof. (Only for *wto*.)

$wto.(skip; S).p.q.r$
$=$ {definition}
$wto.skip.p.q.(wev.S.q.r) \wedge wlp.skip.(wto.S.p.q.r)$
$=$ {definition}
$(\neg p \vee q \vee wev.S.q.r) \wedge wto.S.p.q.r$
$=$ {$e3'$, lemma 72(ii)}
$wto.S.p.q.r$
$=$ {$l2, l1 : wto.S.p.q.r \Rightarrow wlto.S.p.q.(\neg p \vee q \vee r), l5'$}
$wto.S.p.q.(q \vee r) \wedge wlto.S.p.q.(\neg p \vee q \vee r)$
$=$ {definition *skip*, $l0$}
$wto.S.p.q.(wlev.skip.q.r) \wedge wlp.S.(\neg p \vee q \vee r)$
$=$ {definition *skip*}
$wto.S.p.q.(wlev.skip.q.r) \wedge wlp.S.(wto.skip.p.q.r)$
$=$ {definition ";"}
$wto.(S; skip).p.q.r$

□

Theorem 74 *With respect to wlto and wto we have for arbitrary S,*

$$abort; S \;=\; abort.$$

Proof. (Only for *wto*.)

$$\begin{array}{ll} & wto.(abort; S).p.q.r \\ = & \quad \{\text{definition ";'}\} \\ & wto.abort.p.q.(wev.S.q.r) \wedge wlp.abort.(wto.S.p.q.r) \\ = & \quad \{\text{definition } abort\} \\ & p \Rightarrow q \\ = & \quad \{\text{definition } abort\} \\ & wto.abort.p.q.r \end{array}$$

□

Theorem 75 *For program S and predicates p, q, r and w we have,*

$$\begin{array}{ll} (i) & [wlto.S.p.q.r \wedge wlp.S.w \Rightarrow wlto.S.p.q.(r \wedge w)], \\ (ii) & [wlto.S.p.q.r \wedge wp.S.w \Rightarrow wto.S.p.q.(r \wedge w)], \\ (iii) & [wto.S.p.q.r \wedge wlp.S.w \Rightarrow wto.S.p.q.(r \wedge w)], \\ (iv) & [wto.S.p.q.r \wedge wp.S.w \Rightarrow wto.S.p.q.(r \wedge w)]. \end{array}$$

Proof. We give only the proof of (iv); the others are similar.

$$\begin{array}{ll} & wto.S.p.q.r \wedge wp.S.w \\ = & \quad \{l0'\} \\ & wto.S.p.q.r \wedge wto.S.true.false.w \\ \Rightarrow & \quad \{\text{theorem } 70(ii), l4'\} \\ & wto.S.p.q.r \wedge wto.S.p.q.w \\ = & \quad \{l1'\} \\ & wto.S.p.q.(r \wedge w) \end{array}$$

□

The counterpart of theorem 69 does not exactly reduce the proof of $wto.DO.p.q.r$ to proving something about S. It reduces to complicated problem of proving $wto.DO.p.q.r$, to proving something that involves $wev.DO.q.r$.

Theorem 76 *Consider the repetition* **do** $B \rightarrow S$ **od**. *Let J be some predicate. We have,*

$$\begin{array}{l} [J \wedge B \Rightarrow wto.S.p.q.(wev.DO.q.r) \wedge wlp.S.(J \wedge (B \vee \neg p \vee q \vee r))] \\ \Rightarrow \\ [J \wedge (B \vee \neg p \vee q \vee r) \Rightarrow wto.DO.p.q.r]. \end{array}$$

Proof. We employ the fact that *wto* is the weakest solution of $[Y \equiv G.p.q.r.Y]$. We may use (4.3) and (4.4).

$$
\begin{array}{ll}
 & [J \wedge (B \vee \neg p \vee q \vee r) \Rightarrow wto.DO.p.q.r] \\
\Leftarrow & \quad \{(4.4)\} \\
 & [J \wedge (B \vee \neg p \vee q \vee r) \Rightarrow G.p.q.r.(J \wedge (B \vee \neg p \vee q \vee r))] \\
\Leftarrow & \quad \{\text{calculus}\} \\
 & [J \wedge B \Rightarrow G.p.q.r.(J \wedge (B \vee \neg p \vee q \vee r)] \ \wedge \\
 & [(\neg p \vee q \vee r) \wedge \neg B \Rightarrow G.p.q.r.(J \wedge (B \vee \neg p \vee q \vee r))] \\
= & \quad \{\text{definition } G, \text{ calculus}\} \\
 & [J \wedge B \Rightarrow wto.S.p.q.(wev.DO.q.r) \wedge wlp.S.(J \wedge (B \vee \neg p \vee q \vee r))]
\end{array}
$$

□

Our final theorem reduces the proof burden somewhat if we want to prove a *leads-to* property of sequential composition. It is inspired by property 57(*ii*).

Theorem 77 *For programs S and U and predicates p, q and r we have*

$$(i) \quad wlp.S.(wlto.U.p.q.r \wedge wlev.U.q.r) \ \Rightarrow \ wlto.(S;U).p.q.r$$

$$(ii) \quad wp.S.(wto.U.p.q.r \wedge wev.U.q.r) \ \Rightarrow \ wto.(S;U).p.q.r$$

Proof.

$$
\begin{array}{lll}
(i) & & wlp.S.(wlto.U.p.q.r \wedge wlev.U.q.r) \\
 & = & \quad \{wlp \text{ is conjunctive}\} \\
 & & wlp.S.(wlto.U.p.q.r) \wedge wlp.S.(wlev.U.q.r) \\
 & = & \quad \{l0 \text{ for } S\} \\
 & & wlp.S.(wlto.U.p.q.r) \wedge wlto.S.true.false.(wlev.U.q.r) \\
 & \Rightarrow & \quad \{l4 \text{ for } S, \text{ theorem } 70(i)\} \\
 & & wlp.S.(wlto.U.p.q.r) \wedge wlto.S.p.q.(wlev.U.q.r) \\
 & = & \quad \{\text{definition}\} \\
 & & wlto.(S;U).p.q.r \\
(ii) & & wp.S.(wto.U.p.q.r \wedge wev.U.q.r) \\
 & - & \quad \{wp \text{ is conjunctive, } e2 \text{ for } wp\} \\
 & & wlp.S.(wto.U.p.q.r) \wedge wp.S.(wev.U.q.r) \\
 & = & \quad \{l0' \text{ for } S\} \\
 & & wlp.S.(wto.U.p.q.r) \wedge wto.S.true.false.(wev.U.q.r) \\
 & \Rightarrow & \quad \{l4' \text{ for } S, \text{ theorem } 70(ii)\} \\
 & & wlp.S.(wto.U.p.q.r) \wedge wto.S.p.q.(wev.U.q.r) \\
 & = & \quad \{\text{definition}\} \\
 & & wto.(S;U).p.q.r
\end{array}
$$

□

7.3 Proofs of the requirements

Just as in the previous chapter we verify the requirements that we have introduced for all constructs in our language. For the definition of wp and wlp, we refer to the previous chapter. In the following, V is a set of predicates. The requirements that we must verify are the following.

$l0$: $[wlto.S.p.q.(\neg p \vee q \vee r) \equiv wlp.S.(\neg p \vee q \vee r)]$

$l0'$: $[wto.S.true.false.r \equiv wp.S.r]$

$l1$: $wlto.S.p.q$ is universally conjunctive

$l2$: $[wto.S.p.q.r \equiv wto.S.p.q.true \wedge wlto.S.p.q.r]$

$l3'$: $[wto.S.p.p.r]$

$l4$: $[q \Rightarrow q'] \Rightarrow [wlto.S.p.q.r \Rightarrow wlto.S.p.q'.r]$

$l4'$: $[q \Rightarrow q'] \Rightarrow [wto.S.p.q.r \Rightarrow wto.S.p.q'.r]$

$l5$: $[wlto.S.p.q.r \equiv wlto.S.p.q.(q \vee r)]$

$l6$: $[\forall(v : v \in V : wlto.S.v.q.r) \equiv wlto.S.(\exists(v : v \in V : v)).q.r]$

$l6'$: $[\forall(v : v \in V : wto.S.v.q.r) \equiv wto.S.(\exists(v : v \in V : v)).q.r]$

$l7$: $[wlto.S.p.q.r \wedge wlev.S.p.r \Rightarrow wlev.S.q.r]$

$l7'$: $[wto.S.p.q.r \wedge wev.S.p.r \Rightarrow wev.S.q.r]$

$l8$: $[wlto.S.p.q.r \wedge wlto.S.q.w.r \Rightarrow wlto.S.p.w.r]$

$l8'$: $[wto.S.p.q.r \wedge wto.S.q.w.r \Rightarrow wto.S.p.w.r]$

This is really a long list to verify. It reflects the fact that *leads-to.p.q* is a strong property of a program. Many of the proofs are rather straightforward but a couple of them are surprisingly difficult, especially the ones that cope with sequential composition.

7.3.1 *skip*

$l0$: $[wlto.skip.p.q.(\neg p \vee q \vee r) \equiv wlp.skip.(\neg p \vee q \vee r)]$

$=$ {definition}

$[\neg p \vee q \vee r \equiv \neg p \vee q \vee r]$

$l0'$: similar reasoning

$l1$: disjunction with a constant (viz. $\neg p \vee q$) is universally conjunctive

$l2$: follows from $[\neg p \vee q \vee r \equiv (\neg p \vee q \vee true) \wedge (\neg p \vee q \vee r)]$

$l3'$: $[wto.skip.p.p.r]$

$=$ {definition}

$[\neg p \vee p \vee r]$

$l4$: follows from $[q \Rightarrow q'] \Rightarrow [\neg p \vee q \vee r \Rightarrow \neg p \vee q' \vee r]$

$l4'$: same reasoning

$l5$: $[wlto.skip.p.q.r \equiv wlto.skip.p.q.(q \vee r)]$

$=$ {definition}

$[\neg p \vee q \vee r \equiv \neg p \vee q \vee (q \vee r)]$

$l6:$ $[\forall(v : v \in V : wlto.skip.v.q.r) \equiv wlto.skip.(\exists(v : v \in V : v)).q.r]$

$=$ {definition}

$[\forall(v : v \in V : \neg v \vee q \vee r) \equiv \neg\exists(v : v \in V : v) \vee q \vee r]$

$=$ {calculus}

$true$

$l6':$ same reasoning

$l7:$ $[wlto.skip.p.q.r \wedge wlev.skip.p.r \Rightarrow wlev.skip.q.r]$

$=$ {definition}

$[(\neg p \vee q \vee r) \wedge (p \vee r) \Rightarrow (q \vee r)]$

$=$ {calculus}

$true$

$l7':$ same reasoning

$l8:$ $[wlto.skip.p.q.r \wedge wlto.skip.q.w.r \Rightarrow wlto.skip.p.w.r]$

$=$ {definition}

$[(\neg p \vee q \vee r) \wedge (\neg q \vee w \vee r) \Rightarrow (\neg p \vee w \vee r)]$

$=$ {calculus}

$[((\neg p \vee q) \wedge (\neg q \vee w)) \vee r \Rightarrow (\neg p \vee w \vee r)]$

$=$ {calculus}

$true$

$l8':$ same reasoning

7.3.2 *abort*

$l0:$ $[wlto.abort.p.q.(\neg p \vee q \vee r) \equiv wlp.abort.(\neg p \vee q \vee r)]$

$=$ {definition}

$[true \equiv true]$

$l0':$ $[wto.abort.true.false.r \equiv wp.abort.r]$

$=$ {definition}

$[true \Rightarrow false) \equiv false]$

$l1:$ the constant function $true$ is universally conjunctive

$l2:$ $[wto.abort.p.q.r \equiv wto.abort.p.q.true \wedge wlto.abort.p.q.r]$

$=$ {definition}

$[(p \Rightarrow q) \equiv (p \Rightarrow q) \wedge true]$

$l3':$ $[wto.abort.p.p.r]$

$=$ {definition}

$[p \Rightarrow p]$

$l4:$ follows from $[q \Rightarrow q'] \Rightarrow [true \Rightarrow true]$

$l4':$ follows from $[q \Rightarrow q'] \Rightarrow [(p \Rightarrow q) \Rightarrow (p \Rightarrow q')]$

$l5:$ $[wlto.abort.p.q.r \equiv wlto.abort.p.q.(q \vee r)]$

$=$ {definition}

$[true \equiv true]$

$l6:$ $[\forall(v : v \in V : wlto.abort.v.q.r) \equiv wlto.abort.(\exists(v : v \in V : v)).q.r]$

$=$ {definition}

$[\forall(v : v \in V : true) \equiv true]$

$l6':$ $[\forall(v : v \in V : wto.abort.v.q.r) \equiv wto.abort.(\exists(v : v \in V : v)).q.r]$

$=$ {definition}

$[\forall(v : v \in V : v \Rightarrow q) \equiv \exists(v : v \in V : v) \Rightarrow q]$

$=$ {calculus}

$[\forall(v : v \in V : \neg v) \vee q \equiv \neg\exists(v : v \in V : v) \vee q]$

$=$ {calculus}

$true$

$l7:$ $[wlto.abort.p.q.r \wedge wlev.abort.p.r \Rightarrow wlev.abort.q.r]$

$=$ {definition}

$[true \wedge true \Rightarrow true]$

$l7':$ $[wto.abort.p.q.r \wedge wev.abort.p.r \Rightarrow wev.abort.q.r]$

$=$ {definition}

$[(p \Rightarrow q) \wedge p \Rightarrow q]$

$=$ {calculus}

$true$

$l8:$ $[wlto.abort.p.q.r \wedge wlto.abort.q.w.r \Rightarrow wlto.abort.p.w.r]$

$=$ {definition}

$[true \wedge true \Rightarrow true]$

$l8':$ $[wto.abort.p.q.r \wedge wto.abort.q.w.r \Rightarrow wto.abort.p.w.r]$

$=$ {definition}

$[(p \Rightarrow q) \wedge (q \Rightarrow w) \Rightarrow (p \Rightarrow w)]$

$=$ {$\Rightarrow$ is transitive}

$true$

7.3.3 $y := e$

$l0:$ $[wlto.(y := e).p.q.(\neg p \vee q \vee r) \equiv wlp.(y :- e).(\neg p \vee q \vee r)]$

$=$ {definition}

$[(\neg p \vee q \vee q_e^y \vee (\neg p \vee q \vee r)_e^y) \wedge (\neg p_e^y \vee q_e^y \vee (\neg p \vee q \vee r)_e^y) \equiv (\neg p \vee q \vee r)_e^y]$

$=$ {substitution distributes over disjunction, calculus}

$true$

$l0'$: $[wto.(y := e).true.false.r \equiv wp.(y := e).r]$

$=$ {definition}

$[(false \vee false \vee false \vee r_e^y) \wedge (false \vee false \vee r_e^y) \equiv r_e^y]$

$=$ {calculus}

$true$

$l1$: $\forall(v : v \in V : wlto.(y := e).p.q.v)$

$=$ {definition}

$\forall(v : v \in V : (\neg p \vee q \vee q_e^y \vee v_e^y) \wedge (\neg p_e^y \vee q_e^y \vee v_e^y))$

$=$ {distribute universal quantification and disjunction with constant}

$(\neg p \vee q \vee q_e^y \vee \forall(v : v \in V : v_e^y)) \wedge (\neg p_e^y \vee q_e^y \vee \forall(v : v \in V : v_e^y))$

$=$ {substitution distributes over quantification, calculus}

$(\neg p \vee q \vee q_e^y \vee \forall(v : v \in V : v)_e^y) \wedge (\neg p_e^y \vee q_e^y \vee \forall(v : v \in V : v)_e^y)$

$=$ {definition}

$wlto.(y := e).p.q.(\forall(v : v \in V : v))$

$l2$: $[wto.(y := e).p.q.r \equiv wto.(y := e).p.q.true \wedge wlto.(y := e).p.q.r]$

$\Leftarrow$ {calculus}

$[wto.(y := e).p.q.r \equiv wlto.(y := e).p.q.r] \wedge [wto.(y := e).p.q.true]$

$=$ {definition}

$true$

$l3'$: $[wto.(y := e).p.p.r]$

$=$ {definition}

$[(\neg p \vee p \vee p_e^y \vee r_e^y) \wedge (\neg p_e^y \vee p_e^y \vee r_e^y)]$

$=$ {calculus}

$true$

$l4$: $[wlto.(y := e).p.q.r \Rightarrow wlto.(y := e).p.q'.r]$

$=$ {definition}

$[(\neg p \vee q \vee q_e^y \vee r_e^y) \wedge (\neg p_e^y \vee q_e^y \vee r_e^y) \Rightarrow$
$(\neg p \vee q' \vee q'^y_e \vee r_e^y) \wedge (\neg p_e^y \vee q'^y_e \vee r_e^y)]$

$\Leftarrow$ {calculus}

$[q_e^y \Rightarrow q'^y_e] \wedge [q \Rightarrow q']$

$=$ {calculus}

$[q \Rightarrow q']$

$l4'$: same reasoning

$l5$: $wlto.(y := e).p.q.r$

$=$ {definition}

$(\neg p \vee q \vee q_e^y \vee r_e^y) \wedge (\neg p_e^y \vee q_e^y \vee r_e^y)$

$=$ {calculus}

$(\neg p \vee q \vee q_e^y \vee (q_e^y \vee r_e^y)) \wedge (\neg p_e^y \vee q_e^y \vee (q_e^y \vee r_e^y))$

$=$ {definition}

$wlto.(y := e).p.q.(q \vee r)$

$l6$: $\forall(v : v \in V : wlto.(y := e).v.q.r)$

$=$ {definition}

$\forall(v : v \in V : (\neg v \vee q \vee q_e^y \vee r_e^y) \wedge (\neg v_e^y \vee q_e^y \vee r_e^y))$

$=$ {calculus}

$(\neg\exists(v : v \in V : v) \vee q \vee q_e^y \vee r_e^y) \wedge (\neg\exists(v : v \in V : v_e^y) \vee q_e^y \vee r_e^y)$

$=$ {definition}

$wlto.(y := e).(\exists(v : v \in v : V)).q.r$

$l6'$: same reasoning

$l7$: $wlto.(y := e).p.q.r \wedge wlev.(y := e).p.r$

$=$ {definition}

$(\neg p \vee q \vee q_e^y \vee r_e^y) \wedge (\neg p_e^y \vee q_e^y \vee r_e^y) \ \wedge \ (p \vee p_e^y \vee r_e^y)$

$=$ {calculus: isolate r_e^y}

$(\neg p \vee q \vee q_e^y) \wedge (\neg p_e^y \vee q_e^y) \wedge (p \vee p_e^y) \ \vee \ r_e^y$

$=$ {distribute $p \vee p_e^y$, absorbtion}

$((q \vee q_e^y) \wedge (\neg p_e^y \vee q_e^x) \wedge p) \vee ((\neg p \vee q \vee q_e^y) \wedge q_e^y \wedge p_e^y) \vee r_e^y$

$\Rightarrow$ {weaken first term, rewrite second term, distribute r_e^y}

$(q \vee q_e^y \vee r_e^y) \vee (q_e^y \wedge p_e^y \vee r_e^y)$

$=$ {second term implies first term}

$q \vee q_e^y \vee r_e^y$

$=$ {definition}

$wlev.(y := e).q.r$

$l7'$: same reasoning

$l8$: $wlto.(y := e).p.q.r \wedge wlto.(y := e).q.w.r$

$=$ {definition}

$(p \Rightarrow (q \vee q_e^y \vee r_e^y)) \wedge (p_e^y \Rightarrow (q_e^y \vee r_e^y)) \ \wedge$

$(q \Rightarrow (w \vee w_c^y \vee r_c^y)) \wedge (q_c^y \Rightarrow (w_c^y \vee r_c^y))$

$\Rightarrow$ {transitivity of $\Rightarrow$}

$(p \Rightarrow (w \vee w_e^y \vee r_e^y)) \wedge (p_e^y \Rightarrow (w_e^y \vee r_e^y))$

$=$ {definition}

$wlto.(y := e).p.w.r$

$l8'$: same reasoning

7.3.4 $S;U$

$l0$: $wlto.(S;U).p.q.(\neg p \vee q \vee r)$

$=$ {definition}

$wlto.S.p.q.(wlev.U.q.(\neg p \vee q \vee r)) \wedge wlp.S.(wlto.U.p.q.(\neg p \vee q \vee r))$

$=$ {$l0$ for U}

$wlto.S.p.q.(wlev.U.q.(\neg p \vee q \vee r)) \wedge wlp.S.(wlp.U.(\neg p \vee q \vee r))$

$=$ {theorem 75(i)}

$wlto.S.p.q.(wlev.U.q.(\neg p \vee q \vee r) \wedge wlp.U.(\neg p \vee q \vee r)) \wedge wlp.S.(wlp.U.(\neg p \vee q \vee r))$

$=$ {$e0, e4$ for U: $wlp.U.(\neg p \vee q \vee r) \Rightarrow wlev.U.q.(\neg p \vee q \vee r)$}

$wlto.S.p.q.(wlp.U.(\neg p \vee q \vee r)) \wedge wlp.S.(wlp.U.(\neg p \vee q \vee r))$

$=$ {$l0$ for S}

$wlto.S.p.q.(wlp.U.(\neg p \vee q \vee r)) \wedge wlto.S.true.false.(wlp.U.(\neg p \vee q \vee r))$

$=$ {theorem 70(i), $l4$ for S}

$wlp.(S;U).(\neg p \vee q \vee r)$

$l0'$: $wto.(S;U).true.false.r$

$=$ {definition}

$wto.S.true.false.(wev.U.false.r) \wedge wlp.S.(wto.U.true.false.r)$

$=$ {$e0'$ for U, $l0'$ for U}

$wto.S.true.false.(wp.U.r) \wedge wlp.S.(wp.U.r)$

$=$ {$l0'$ for S}

$wp.S.(wp.U.r) \wedge wlp.S.(wp.U.r)$

$=$ {definition, first term implies second term}

$wp.(S;U).r$

$l1$: $\forall(v : v \in V : wlto.(S;U).p.q.v)$

$=$ {definition}

$\forall(v : v \in V : wlto.S.p.q.(wlev.U.q.v) \wedge wlp.S.(wlto.U.p.q.v))$

$=$ {distribute quantification, $l1$ for S, $e1$ for U (wlp)}

$wlto.S.p.q.(\forall(v : v \in V : wlev.U.q.v)) \wedge wlp.S.(\forall(v : v \in V : wlto.U.p.q.v))$

$=$ {$e1$ for U, $l1$ for U}

$wlto.S.p.q.(wlev.U.q.(\forall(v : v \in V : v))) \wedge wlp.S.(wlto.U.p.q.(\forall(v : v \in V : v)))$

$=$ {definition}

$wlto.(S;U).p.q.(\forall(v : v \in V : v))$

$l2$: $wto.(S;U).p.q.r$

$=$ {definition}

$wto.S.p.q.(wev.U.q.r) \wedge wlp.S.(wto.U.p.q.r)$

$=$ {$l2$ for S and U, $e2$ for U}

$wlto.S.p.q.(wlev.U.q.r \wedge wev.U.q.true) \wedge wto.S.p.q.true \wedge$
$wlp.S.(wlto.U.p.q.r \wedge wto.U.p.q.true)$

$=$ {$l1$ for S, $e1$ for S (wlp)}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlto.S.p.q.(wev.U.q.true) \wedge wto.S.p.q.true \wedge$
$wlp.S.(wlto.U.p.q.r) \wedge wlp.S.(wto.U.p.q.true)$

$=$ {$l2$ for S, definition}

$wlto.(S;U).p.q.r \wedge wto.S.p.q.(wev.U.q.true) \wedge wlp.S.(wto.U.p.q.true)$

$=$ {definition}

$wlto.(S;U).p.q.r \wedge wto.(S;U).p.q.true$

$l3'$: $[wto.(S;U).p.p.r]$

$=$ {definition}

$[wto.S.p.p.(wev.U.p.r) \wedge wlp.S.(wto.U.p.p.r)]$

$=$ {distribute quantification}

$[wto.S.p.p.(wev.U.p.r)] \wedge [wlp.S.(wto.U.p.p.r)]$

$=$ {$l3'$ for S and U}

$[wlp.S.true]$

$=$ {$e1$}

$true$

$l4$: $[wlto.(S;U).p.q.r \Rightarrow wlto.(S;U).p.q'.r]$

$=$ {definition}

$[wlto.S.p.q.(wlev.U.q.r) \wedge wlp.S.(wlto.U.p.q.r) \Rightarrow$
$wlto.S.p.q'.(wlev.U.q'.r) \wedge wlp.S.(wlto.U.p.q'.r)]$

$\Leftarrow$ {distribute universal quantification}

$[wlto.S.p.q.(wlev.U.q.r) \Rightarrow wlto.S.p.q'.(wlev.U.q'.r)] \wedge$
$[wlp.S.(wlto.U.p.q.r) \Rightarrow wlp.S.(wlto.U.p.q'.r)]$

$\Leftarrow$ {$wlp.S$ monotonic, $wlto.S.p.q$ monotonic}

$[wlto.S.p.q.(wlev.U.q.r) \Rightarrow wlto.S.p.q'.(wlev.U.q.r)] \wedge$
$[wlev.U.q.r \Rightarrow wlev.U.q'.r] \wedge [wlto.U.p.q.r \Rightarrow wlto.U.p.q'.r]$

$\Leftarrow$ {$l4$ for S and U, $e4$ for S}

$[q \Rightarrow q']$

$l4'$: similar reasoning

$l5$: $wlto.(S;U).p.q.r$

$=$ {definition}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlp.S.(wlto.U.p.q.r)$

$=$ {$l5$ and $e5$ for U}

$wlto.S.p.q.(wlev.U.q.(q \vee r)) \wedge wlp.S.(wlto.U.p.q.(q \vee r))$

$=$ {definition}

$wlto.(S;U).p.q.(q \vee r)$

$l6:$ $\forall(v : v \in V : wlto.(S;U).v.q.r)$

$=$ {definition}

$\forall(v : v \in V : wlto.S.v.q.(wlev.U.q.r) \wedge wlp.S.(wlto.U.v.q.r))$

$=$ {calculus, $wlp.S$ universally conjunctive}

$\forall(v : v \in V : wlto.S.v.q.(wlev.U.q.r)) \wedge wlp.S.(\forall(v : v \in V : wlto.U.v.q.r))$

$=$ {$l6$ for S and U}

$wlto.S.(\exists(v : v \in V : v)).q.(wlev.U.q.r) \wedge wlp.S.(wlto.U.(\exists(v : v \in V : v)).q.r)$

$=$ {definition}

$wlto.(S;U).(\exists(v : v \in V : v)).q.r$

$l6':$ similar reasoning

$l7:$ $wlto.(S;U).p.q.r \wedge wlev.(S;U).p.r$

$=$ {definition}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlp.S.(wlto.U.p.q.r) \wedge wlev.S.p.(wlev.U.p.r)$

$\rightarrow$ {theorem $68(i)$}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlev.S.p.(wlto.U.p.q.r \wedge wlev.U.p.r)$

$\Rightarrow$ {$l7$ for U, $wlev.S.q$ monotonic}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlev.S.p.(wlev.U.q.r)$

$\Rightarrow$ {$l7$ for S}

$wlev.S.q.(wlev.U.q.r)$

$=$ {definition}

$wlev.(S;U).q.r$

$l7':$ similar reasoning

$l8:$ $wlto.(S;U).p.q.r \wedge wlto.(S;U).q.w.r$

$=$ {definition}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlto.S.q.w.(wlev.U.w.r) \wedge$
$wlp.S.(wlto.U.p.q.r) \wedge wlp.S.(wlto.U.q.w.r)$

$\Rightarrow$ {wlp conjunctive, $l8$ for U}

$wlto.S.p.q.(wlev.U.q.r) \wedge wlto.S.q.w.(wlev.U.w.r) \wedge wlp.S.(wlto.U.p.w.r)$

$=$ {theorem $75(i)$}

$wlto.S.p.q.(wlev.U.q.r \wedge wlto.U.q.w.r) \wedge wlto.S.q.w.(wlev.U.w.r) \wedge$
$wlp.S.(wlto.U.p.w.r)$

$\Rightarrow$ {$l7$ for U, $wlto.S.p.q$ monotonic}

$wlto.S.p.q.(wlev.U.w.r) \wedge wlto.S.q.w.(wlev.U.w.r) \wedge wlp.S.(wlto.U.p.w.r)$

$\Rightarrow$ {$l8$ for S}

$wlto.S.p.w.(wlev.U.w.r) \wedge wlp.S.(wlto.U.p.w.r)$

$=$ {definition}

$wlto.(S;U).p.w.r$

$l8'$: similar reasoning

7.3.5 if $[](i :: B_i \rightarrow S_i)$ fi

We abbreviate **if** $[](i :: B_i \rightarrow S_i)$ **fi** by *IF*.

$l0$: $wlto.IF.p.q.(\neg p \vee q \vee r)$

$=$ {definition}

$\forall(i : B_i : wlto.S_i.p.q.(\neg p \vee q \vee r))$

$=$ {$l0$ for the S_i}

$\forall(i : B_i : wlp.S_i.(\neg p \vee q \vee r))$

$=$ {definition}

$wlp.IF.(\neg p \vee q \vee r)$

$l0'$: $wto.IF.true.false.r$

$=$ {definition}

$(true \Rightarrow false \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.true.false.r)$

$-$ {$l0'$ for the S_i}

$\exists(i :: B_i) \wedge \forall(i : B_i : wp.S_i.r)$

$=$ {definition}

$wp.IF.r$

$l1$: $\forall(v : v \in V : wlto.IF.p.q.v)$

$-$ {definition}

$\forall(v : v \in V : \forall(i : B_i : wlto.S_i.p.q.v))$

$=$ {interchange universal quantifications}

$\forall(i : B_i : \forall(v : v \in V : wlto.S_i.p.q.v))$

$=$ {$l1$ for the S_i}

$\forall(i : B_i : wlto.S_i.p.q.(\forall(v : v \in V : v)))$

$=$ {definition}

$wlto.IF.p.q.(\forall(v : v \in V : v))$

$l2$: $wto.IF.p.q.r$

$=$ {definition}

$((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q.r)$

$=$ $\{l2$ for the S_i, calculus$\}$

$((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q.true) \wedge \forall(i : B_i : wlto.S_i.p.q.r)$

$=$ {definition}

$wto.IF.p.q.true \wedge wlto.IF.p.q.r$

$l3'$: $[wto.IF.p.p.r]$

$=$ {definition}

$[((p \Rightarrow p) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.p.r)]$

$=$ $\{l3'$ for the $S_i\}$

$true$

$l4$: similar to, even easier than, $l4'$

$l4'$: $[wto.IF.p.q.r \Rightarrow wto.IF.p.q'.r]$

$=$ {definition}

$[((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q.r) \Rightarrow$
$((p \Rightarrow q') \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q'.r)]$

$\Leftarrow$ {calculus, $l4'$ for the S_i}

$[q \Rightarrow q']$

$l5$: $wlto.IF.p.q.r$

$=$ {definition}

$\forall(i : B_i : wlto.S_i.p.q.r)$

$=$ $\{l5$ for the $S_i\}$

$\forall(i : B_i : wlto.S_i.p.q.(q \vee r))$

$=$ {definition}

$wlto.IF.p.q.(q \vee r)$

$l6$: similar to $l6'$

$l6'$: $\forall(v : v \in V : wto.IF.v.q.r)$

$=$ {definition}

$\forall(v : v \in V : ((v \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.v.q.r))$

$=$ {calculus}

$\forall(v : v \in V : ((v \Rightarrow q) \vee \exists(i :: B_i))) \wedge \forall(v : v \in V : \forall(i : B_i : wto.S_i.v.q.r))$

$=$ $\{$calculus, $l6'$ for the $S_i\}$

$((\exists(v : v \in V : v) \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.(\exists(v : v \in V : v)).q.r)$

$=$ {definition}

$wto.IF.(\exists(v : v \in V : v)).q.r$

$l7$: similar to $l7'$

$l7'$: $wto.IF.p.q.r \wedge wev.IF.p.r$

$=$ {definition}

$$
\begin{array}{ll}
 & ((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q.r) \wedge \\
 & \qquad (p \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wev.S_i.p.r) \\
\Rightarrow & \quad \{\text{calculus, } l7' \text{ for the } S_i\} \\
 & ((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge (p \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wev.S_i.q.r) \\
= & \quad \{\text{distribute } p \vee \exists(i :: B_i)\} \\
 & (p \wedge ((p \Rightarrow q) \vee \exists(i :: B_i)) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wev.S_i.q.r) \\
\Rightarrow & \quad \{\text{calculus}\} \\
 & (q \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wev.S_i.q.r) \\
= & \quad \{\text{definition}\} \\
 & wev.IF.q.r \\
l8: & \quad \text{similar to } l8' \\
l8': & \quad wto.IF.p.q.r \wedge wto.IF.q.w.r \\
= & \quad \{\text{definition}\} \\
 & ((p \Rightarrow q) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.q.r) \wedge \\
 & \qquad ((q \Rightarrow w) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.q.w.r) \\
\Rightarrow & \quad \{l8' \text{ for the } S_i, \text{ transitivity of } \Rightarrow\} \\
 & ((p \Rightarrow w) \vee \exists(i :: B_i)) \wedge \forall(i : B_i : wto.S_i.p.w.r) \\
= & \quad \{\text{definition}\} \\
 & wto.IF.p.w.r
\end{array}
$$

7.3.6 do $B \rightarrow S$ od

Again the proofs for **do** $B \rightarrow S$ **od**, (abbreviated by DO), are completely different from the ones given above. They turn out to be quite complicated in spite of the fact that the equations for $wlto$ and wto are a lot simpler than the equations for $wlev$ and wev (although this may not be clear at first sight). Both equations ((7.1) and (7.2)) have the general form

$$[Y \equiv (\neg B \vee wlp.S.Y) \wedge z]. \tag{7.3}$$

We call the right-hand side of this equation $H.z.Y$. Again, we call the weakest solution of the equation, $h.z$ and the strongest $g.z$. In this way we have

$$wlto.DO.p.q.r = h.((\neg B \vee wlto.S.p.q.(wlev.DO.q.r)) \wedge (B \vee \neg p \vee q \vee r)), \tag{7.4}$$

$$wto.DO.p.q.r = h.((\neg B \vee wto.S.p.q.(wev.DO.q.r)) \wedge (B \vee \neg p \vee q \vee r)). \tag{7.5}$$

We first prove some facts about h. Since H is $\{0,1\}$-monotonic, h is monotonic. We even have h is universally conjunctive since H is $\{0,1\}$-universally conjunctive. To prove $l0$ we need the following lemma.

Lemma 78 *For fixed predicate z,*

- *(i) The conjunction of every two solutions of (7.3) is a solution of (7.3),*
- *(ii) For x a solution of (7.3), $h.z \wedge x$ is the weakest solution of $[Y \equiv H.z.(x \wedge Y)]$.*

Proof. (i) Let x and y be solutions of (7.3).

$$
\begin{array}{ll}
 & [x \wedge y = H.z.(x \wedge y)] \\
= & \quad \{z \equiv z \wedge z, H \ \{0,1\}\text{-conjunctive}\} \\
 & [x \wedge y \equiv H.z.x \wedge H.z.y] \\
= & \quad \{x \text{ and } y \text{ are solutions}\} \\
 & true
\end{array}
$$

(ii) Let x be a solution of (7.3). According to (i), $h.z \wedge x$ is a solution of $[Y \equiv H.z.(x \wedge Y)]$. We can rewrite the latter equation using the fact that x solves (7.3).

$$
\begin{array}{ll}
 & H.z.(x \wedge Y) \\
= & \quad \{H\{0,1\}\text{-conjunctive}\} \\
 & H.z.x \wedge H.z.Y \\
= & \quad \{x \text{ solves } 7.3\} \\
 & x \wedge H.z.Y
\end{array}
$$

It follows that we can rewrite the equation $[Y \equiv H.z.(x \wedge Y)]$ as $[Y \equiv x \wedge H.z.Y]$. The weakest solution of this equation is $h.(x \wedge z)$. Now,

$$
\begin{array}{ll}
 & h.(x \wedge z) \\
= & \quad \{h \text{ is conjunctive}\} \\
 & h.x \wedge h.z \\
\Rightarrow & \quad \{h.x = x \wedge (\neg B \vee wlp.S.(h.x))\} \\
 & x \wedge h.z.
\end{array}
$$

Since the weakest solution implies another solution, those two solutions are the same.

□

Notice that $wlp.DO.r = h.(B \vee r)$. We use this to prove $l0$.

$$
\begin{array}{ll}
 & wlto.DO.p.q.r \\
= & \quad \{(7.1)\} \\
 & h.((\neg B \vee wlto.S.p.q.(wlev.DO.q.r)) \wedge (B \vee \neg p \vee q \vee r)) \\
\Rightarrow & \quad \{h \text{ monotonic}\} \\
 & h.(B \vee \neg p \vee q \vee r) \\
= & \quad \{\text{definition}\} \\
 & wlp.DO.(\neg p \vee q \vee r)
\end{array}
$$

This gives us one implication of the equivalence. We now prove

$$[wlto.DO.true.false.r \equiv wlp.DO.r].$$

We then have

$$wlp.DO.(\neg p \vee q \vee r)$$

$$
\begin{array}{ll}
= & \\
 & wlto.DO.true.false.(\neg p \vee q \vee r) \\
\Rightarrow & \quad \{l4, l6\} \\
 & wlto.DO.p.q.(\neg p \vee q \vee r)
\end{array}
$$

which is the other part of the equivalence. We have to see to it that we do not use $l0$ in the proofs of $l4$ and $l6$.

$$
\begin{array}{ll}
 & wlto.DO.true.false.r \\
= & \quad \{(7.4)\} \\
 & h.((\neg B \vee wlto.S.true.false.(wlev.DO.false.r)) \wedge (B \vee r)) \\
= & \quad \{l0 \text{ for } S,\ e0 \text{ for } DO\} \\
 & h.((\neg B \vee wlp.S.(wlp.DO.r)) \wedge (B \vee r)) \\
= & \quad \{wlp \text{ conjunctive, lemma 78 with } x := wlp.DO.r, z := B \vee r\} \\
 & h.(B \vee r) \wedge wlp.DO.r \\
= & \quad \{\text{same terms}\} \\
 & wlp.DO.r
\end{array}
$$

The proof of $l0'$ is similar (complicated). Notice, $wp.DO.r = g.((\neg B \vee wp.S.true) \wedge (B \vee r))$.

$$
\begin{array}{ll}
 & wto.DO.true.false.r \\
= & \quad \{(7.5)\} \\
 & h.((\neg B \vee wto.S.true.false.(wev.DO.false.r)) \wedge (B \vee r)) \\
= & \quad \{l1' \text{ for } S,\ e0' \text{ for } DO\} \\
 & h.((\neg B \vee wp.S.(wp.DO.r)) \wedge (B \vee r)) \\
= & \quad \{wp.S.r = wp.S.true \wedge wlp.S.r\} \\
 & h.((\neg B \vee wp.S.true) \wedge (\neg B \vee wlp.S.(wp.DO.r)) \wedge (B \vee r)) \\
= & \quad \{\text{lemma 78 with } z := (\neg B \vee wp.S.true) \wedge (B \vee r), x := wp.DO.r\} \\
 & h.((\neg B \vee wp.S.true) \wedge (B \vee r)) \wedge wp.DO.r \\
= & \quad \{\text{same terms}\} \\
 & wp.DO.r
\end{array}
$$

These proofs are surprising in two respects. Firstly, they are rather difficult whereas this were not the most difficult requirements to prove for the other constructs. Secondly, for $l0'$, we started out with a weakest solution of equation (7.3) and ended with a strongest solution.

We already have that h is universally conjunctive. $l1$ now follows from $l1$ for S and from the fact that disjunction with a constant distributes over universal quantification. We proceed with $l2$.

$$
\begin{array}{ll}
 & wto.DO.p.q.r \\
= & \quad \{(7.5)\}
\end{array}
$$

$$
\begin{array}{ll}
 & h.((\neg B \vee wto.S.p.q.(wev.DO.q.r)) \wedge (B \vee \neg p \vee q \vee r)) \\
= & \quad \{e2 \text{ for } DO,\ l2 \text{ for } S\} \\
 & h.((\neg B \vee wlto.S.p.q.(wev.DO.q.true \wedge wlev.DO.q.r) \wedge wto.S.p.q.true) \wedge \\
 & \qquad (B \vee \neg p \vee q \vee r)) \\
= & \quad \{l1 \text{ for } S\} \\
 & h.((\neg B \vee wlto.S.p.q.(wlev.DO.q.r) \wedge wlto.S.p.q.(wev.DO.q.true) \wedge wto.S.p.q.true) \wedge \\
 & \qquad (B \vee \neg p \vee q \vee r)) \\
= & \quad \{l2 \text{ for } S\} \\
 & h.((\neg B \vee wlto.S.p.q.(wlev.DO.q.r) \wedge wto.S.p.q.(wev.DO.q.true)) \wedge (B \vee \neg p \vee q \vee r)) \\
= & \quad \{h \text{ conjunctive}\} \\
 & h.((\neg B \vee wlto.S.p.q.(wlev.DO.q.r)) \wedge (B \vee \neg p \vee q \vee r)) \wedge \\
 & \qquad h.((\neg B \vee wto.S.p.q.(wev.DO.q.true)) \wedge (B \vee \neg p \vee q \vee r)) \\
= & \quad \{(7.4), (7.5)\} \\
 & wlto.DO.p.q.r \wedge wto.DO.p.q.true
\end{array}
$$

$l3'$ is next.

$$
\begin{array}{ll}
 & [wto.DO.p.p.r] \\
= & \quad \{(7.5)\} \\
 & [h.((\neg B \vee wto.S.p.p.(wev.DO.p.r)) \wedge (B \vee \neg p \vee p \vee r))] \\
= & \quad \{l3' \text{ for } S\} \\
 & [h.true] \\
= & \quad \{h \text{ universally conjunctive}\} \\
 & true
\end{array}
$$

$l4$ and $l4'$ are also straightforward. From the requirements for which *wlto* and *wto* have similar proofs, we tackle only one and leave the other one to the reader. We prove $l4'$.

$$
\begin{array}{ll}
 & [wto.DO.p.q.r \Rightarrow wto.DO.p.q'.r] \\
= & \quad \{(7.4)\} \\
 & [h.((\neg B \vee wto.S.p.q.(wev.DO.q.r)) \wedge (B \vee \neg p \vee q \vee r)) \Rightarrow \\
 & \qquad h.((\neg B \vee wto.S.p.q'.(wev.DO.q'.r)) \wedge (B \vee \neg p \vee q' \vee r))] \\
\Leftarrow & \quad \{h \text{ monotonic, calculus}\} \\
 & [wto.S.p.q.(wev.DO.q.r) \Rightarrow wto.S.p.q'.(wev.DO.q'.r)] \wedge [q \Rightarrow q'] \\
\Leftarrow & \quad \{e4' \text{ for } DO,\ wto.S.p.q \text{ monotonic}\} \\
 & [wto.S.p.q.(wev.DO.q.r) \Rightarrow wto.S.p.q'.(wev.DO.q.r)] \wedge [q \Rightarrow q'] \\
= & \quad \{l4' \text{ for } S\} \\
 & [q \Rightarrow q']
\end{array}
$$

$l5$ is easy: using $l5$ for S we observe that the defining equation for $wlto.DO.p.q.r$ does not change if we substitute $q \vee r$ for r. For $l6$ observe,

$$
\begin{array}{ll}
 & \forall(v : v \in V : wlto.DO.v.q.r) \\
= & \quad \{(7.4)\} \\
 & \forall(v : v \in V : h.((\neg B \vee wlto.S.v.q.(wlev.DO.q.r)) \wedge (B \vee \neg v \vee q \vee r))) \\
= & \quad \{h \text{ is universally conjunctive}\} \\
 & h.(\forall(v : v \in V : (\neg B \vee wlto.S.v.q.(wlev.DO.q.r)) \wedge (B \vee \neg v \vee q \vee r))) \\
 & \quad \{\text{calculus, } l6 \text{ for } S\} \\
 & h.((\neg B \vee wlto.S.(\exists(v : v \in V : v)).q.(wlev.DO.q.r)) \wedge (B \vee \neg \exists(v : v \in V : v) \vee q \vee r))) \\
= & \quad \{(7.4)\} \\
 & wlto.DO.(\exists(v : v \in V : v)).q.r.
\end{array}
$$

In the following proofs, we employ explicitly that $wlto$, wto, $wlev$ and wev are extreme solutions, i.e., we use characterizations (4.1) through (4.4). We want to use (4.4) in the proof of $l7$. When we apply it directly, it turns out that we have to prove

$$
\begin{array}{l}
[wlto.DO.p.q.r \wedge wlev.DO.p.r \Rightarrow \\
\qquad (\neg B \vee wlev.S.q.(wlto.DO.p.q.r \wedge wlev.DO.p.r)) \wedge (B \vee q \vee r)].
\end{array}
$$

This is too strong to hope for. Operationally speaking, the possibility of p becoming *true* only in the first iteration and q becoming *true* only after the first iteration is ruled out in the right-hand side. We therefore look for an intermediate predicate w such that

$$
\begin{array}{ll}
(i) & [wlto.DO.p.q.r \wedge wlev.DO.p.r \Rightarrow w], \\
(ii) & [w \Rightarrow (\neg B \vee wlev.S.q.w) \wedge (B \vee q \vee r)].
\end{array}
$$

By using (4.4), the transitivity of the implication and (6.1), $l7$ follows. We propose $w = wlto.DO.p.q.r \wedge wlev.DO.(p \vee q).r$.

$$
\begin{array}{lll}
(i) & & \text{follows from } e4 \text{ for } DO \\
(ii) & & wlto.DO.p.q.r \wedge wlev.DO.(p \vee q).r \\
 & = & \quad \{(7.1), (6.1)\} \\
 & & (\neg B \vee wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.(wlto.DO.p.q.r) \wedge \\
 & & \qquad wlev.S.(p \vee q).(wlev.DO.(p \vee q).r)) \wedge (B \vee \neg p \vee q \vee r) \wedge (B \vee p \vee q \vee r)
\end{array}
$$

Since $(B \vee \neg p \vee q \vee r) \wedge (B \vee p \vee q \vee r) \equiv (B \vee q \vee r)$, our proof obligation boils down to proving

$$
\begin{array}{l}
[wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.(wlto.DO.p.q.r) \wedge \\
\qquad wlev.S.(p \vee q).(wlev.DO.(p \vee q).r) \\
\Rightarrow \\
\quad wlev.S.q.(wlto.DO.p.q.r \wedge wlev.DO.(p \vee q).r)].
\end{array}
$$

We use $l3 : [wlto.S.q.q.(wlev.DO.(p \vee q).r)]$.

$$
\begin{array}{ll}
 & wlto.S.p.q.(wlev.DO.q.r) \wedge wlp.S.(wlto.DO.p.q.r) \wedge \\
 & \qquad wlev.S.(p \vee q).(wlev.DO.(p \vee q).r) \\
\Rightarrow & \quad \{e4 \text{ for } DO, \text{ monotonicity of } wlto.S.p.q, \ l3\} \\
 & wlto.S.p.q.(wlev.DO.(p \vee q).r) \wedge wlto.S.q.q.(wlev.DO.(p \vee q).r) \wedge \\
 & \qquad wlp.S.(wlto.DO.p.q.r) \wedge wlev.S.(p \vee q).(wlev.DO.(p \vee q).r) \\
= & \quad \{l6 \text{ for } S\} \\
 & wlto.S.(p \vee q).q.(wlev.DO.(p \vee q).r) \wedge wlp.S.(wlto.DO.p.q.r) \wedge \\
 & \qquad wlev.S.(p \vee q).(wlev.DO.(p \vee q).r) \\
\Rightarrow & \quad \{l7 \text{ for } S\} \\
 & wlev.S.q.(wlev.DO.(p \vee q).r) \wedge wlp.S.(wlto.DO.p.q.r) \\
\Rightarrow & \quad \{\text{theorem } 68(i)\} \\
 & wlev.S.q.(wlev.DO.(p \vee q).r \wedge wlto.DO.p.q.r)
\end{array}
$$

This proves $l7$. Since $wev.DO.q.r$ is the strongest solution of its defining equation, the proof of $l7'$ is going to be completely different. The strongest solution is characterized by (4.1) and (4.2). We employ (4.2) and in order to do so we rewrite our proof obligation.

$$[wev.DO.p.r \Rightarrow \neg wto.DO.p.q.r \vee wev.DO.q.r]$$

Using (4.2), this follows from

$$
\begin{array}{ll}
 & [(\neg B \vee wev.S.p.(\neg wto.DO.p.q.r \vee wev.DO.q.r)) \wedge (B \vee p \vee r) \Rightarrow \\
 & \qquad \neg wto.DO.p.q.r \vee wev.DO.q.r] \\
= & \quad \{\text{calculus}\} \\
 & [(\neg B \vee wev.S.p.(\neg wto.DO.p.q.r \vee wev.DO.q.r)) \wedge (B \vee p \vee r) \wedge wto.DO.p.q.r \Rightarrow \\
 & \qquad wev.DO.q.r] \\
= & \quad \{(7.2)\} \\
 & [(\neg B \vee wev.S.p.(\neg wto.DO.p.q.r \vee wev.DO.q.r) \wedge wto.S.p.q.(wev.DO.q.r) \wedge \\
 & \qquad wlp.S.(wto.DO.p.q.r)) \wedge (B \vee \neg p \vee q \vee r) \wedge (B \vee p \vee r) \Rightarrow wev.DO.q.r] \\
= & \quad \{\text{theorem } 68(iii), \text{ calculus}\} \\
 & [(\neg B \vee wev.S.p.(wev.DO.q.r) \wedge wto.S.p.q.(wev.DO.q.r) \wedge wlp.S.(wto.DO.p.q.r)) \wedge \\
 & \qquad (B \vee p \wedge q \vee r) \Rightarrow wev.DO.q.r] \\
\Leftarrow & \quad \{l7' \text{ for } S, \text{ strenghtening antecedent}\} \\
 & [(\neg B \vee wev.S.q.(wev.DO.q.r)) \wedge (B \vee q \vee r) \Rightarrow wev.DO.q.r] \\
= & \quad \{\text{definition}\} \\
 & true.
\end{array}
$$

The proofs for $l8$ and $l8'$ are similar again. We prove $l8$.

$$wlto.DO.p.q.r \wedge wlto.DO.q.w.r$$

$=$ {definition}

$(\neg B \vee wlto.S.p.q.(wlev.DO.q.r) \wedge wlto.S.q.w.(wlev.DO.w.r) \wedge$
$\quad wlp.S.(wlto.DO.p.q.r) \wedge wlp.S.(wlto.DO.q.w.r)) \wedge$
$(B \vee \neg p \vee q \vee r) \wedge (B \vee \neg q \vee w \vee r)$

$\Rightarrow$ {theorem 75(i), calculus}

$(\neg B \vee wlto.S.p.q.(wlev.DO.q.r \wedge wlto.DO.q.w.r) \wedge wlto.S.q.w.(wlev.DO.w.r) \wedge$
$\quad wlp.S.(wlto.DO.p.q.r \wedge wlto.DO.q.w.r)) \wedge (B \vee \neg p \vee w \vee r)$

$\Rightarrow$ {$l7$ for S, monotonicity}

$(\neg B \vee wlto.S.p.q.(wlev.DO.w.r) \wedge wlto.S.q.w.(wlev.DO.w.r) \wedge$
$\quad wlp.S.(wlto.DO.p.q.r \wedge wlto.DO.q.w.r)) \wedge (B \vee \neg p \vee w \vee r)$

$\Rightarrow$ {$l8$ for S}

$(\neg B \vee wlto.S.p.w.(wlev.DO.w.r) \wedge wlp.S.(wlto.DO.p.q.r \wedge wlto.DO.q.w.r)) \wedge$
$\quad (B \vee \neg p \vee w \vee r)$

Using characterization (4.4), the result now follows.

Chapter 8

Concluding remarks and further research

The theory, presented in the preceding chapters is a basis for the definition of arbitrary properties of programs. We have shown how the familiar concepts of *wlp* and *wp* are embedded in this theory and we have given two examples of properties that were defined by application of the theory. The reason that we investigated these two properties in so much detail is that they were the starting point for this research. Our interest in especially those two properties originates from our experience in developing parallel programs. For instance, the property *p leads-to q* shows up in parallel programs as follows. Consider the communication statements on a channel *c*. If a process executes $c?y$, it waits until another process performs $c!expr$. In order for this to happen eventually, the process executing $c?y$ must start this communication in a state such that eventually $\overline{c!}$ becomes *true*. In other words, this state must satisfy the weakest predicate *p* such that *p leads-to* $\overline{c!}$. This has been our motivation to look at the property *p leads-to q*. When investigating this property, it turned out that we had to introduce *ever.q* as well.

The proofs of correctness of parallel programs, if given at all, are usually ad hoc and rely on completely different arguments than proofs for sequential programs usually do. Furthermore, although it is usually clear what we want a whole program to do, this does not help us in specifying what a process, i.e., a part of the program, should do. What we hope and aim for is that these new properties help us in specifying and deriving parts of a program. An example of such a specification can be found in [29].

Our aim was to describe the semantics of a programming language containing sequential as well as parallel composition in such a way, that properties of programs can be derived in a compositional way from the program text. We are not that far yet since we did not include parallel composition in the language. In the remainder of this chapter, we show how parallelism can be included and we point out some of the complications.

First we note that, although we do not have parallelism in the language, we can write similar programs as, for instance, can be written in UNITY ([6]). In order to see this we must have a closer look at UNITY. A UNITY program is a finite set of conditional assignments $\{i : i \in I : A_i \textbf{ if } c_i\}$. An execution of a UNITY program is an infinite iteration in which in every

step of the iteration an assignment is selected and, if the condition for that assignment holds, is executed. There is a fairness constraint: in this infinite iteration, every assignment is selected an infinite number of times. Except for this fairness constraint, the UNITY program can be written as

$$\textbf{do } \mathit{true} \rightarrow \textbf{if } [](i : i \in I : c_i \rightarrow A_i) \textbf{ fi od.}$$

The fairness constraint may be incorporated by adding some variables and modifying the guards a little bit but this is not our concern now. The question arises whether there is already some parallelism in our language or, if not, if there exists any parallelism in UNITY. To answer this we look at the intended meaning of $S \parallel T$. In terms of computations, the meaning of $S \parallel T$ corresponds to executing S and T in some interleaved fashion. We can make this interleaving explicit by writing an alternative construct with only "atomic" (indivisible) constructs as possible selections and by allowing multiple assignments. As an example, consider the programs $x := 3; y := 4$ and $x := 5; y := 2$. The parallel execution of these programs consists of all interleavings of the statements in the programs such that the order of the statements within each program is preserved. We can write this as the following program in which we use two fresh variables, $pc0$ and $pc1$.

$$\begin{array}{l} pc0 := 0; pc1 := 0; \\ \textbf{do } pc0 \neq 2 \vee pc1 \neq 2 \rightarrow \textbf{if } pc0 = 0 \rightarrow x, pc0 := 3, 1 \\ \qquad\qquad\qquad\qquad\quad [] \ pc0 = 1 \rightarrow y, pc0 := 4, 2 \\ \qquad\qquad\qquad\qquad\quad [] \ pc1 = 0 \rightarrow x, pc1 := 5, 1 \\ \qquad\qquad\qquad\qquad\quad [] \ pc1 = 1 \rightarrow y, pc1 := 2, 2 \\ \qquad\qquad\qquad\qquad\quad \textbf{fi} \\ \textbf{od} \end{array}$$

In this example, we make in fact the (anonymous) program counters of the programs explicit in order to get rid of the sequential composition. The answer to the above question is: yes, we already have the power to express parallelism in our language by using the nondeterministic alternative construct. We can define the parallel composition of two programs written in this way (i.e., as a repetition of an alternative construct and not containing a semicolon), by constructing a repetition of a new alternative construct which is the union of the two alternative constructs. It has been shown that in this way we can describe and derive reasonable complicated parallel programs ([6, 2, 3, 25]).

Ultimately, we want to reason about programs like the one described in chapter three. We have seen that, if we are willing to rewrite such a program as one alternative statement, we can do it. However, this is not very attractive since we loose the nice compositional aspects associated with a program consisting of communicating processes. It would be nice if we can infer properties from the system as a whole from properties of its components. Before we can do that, we need a definition of the parallel construct. Therefore, we point out a way to include the parallel construct in the operational semantics of chapter five.

What exactly is the problem with the parallel construct? The other constructs in the language share the nice property of compositionality. This means that the meaning of a construct

can be defined entirely in terms of the meaning of the constructs from which it is built. For instance, if we know the meaning of programs S and U we can describe the meaning of $S;U$. The parallel construct does not have this nice property. Suppose we have the following two programs.

$$\begin{array}{ll} S: & x := 0;\ \textbf{if}\ x < 0 \to y := 1\ []\ x \geq 0 \to \mathit{skip}\ \textbf{fi} \\ U: & x := -1 \end{array}$$

In the operational semantics of S, the **if**-statement reduces to *skip* since, in isolation, "$x < 0$" will be *false* after execution of $x := 0$. The problem is that, in the operational semantics, the information that in the execution of S actually a choice was involved, has disappeared. This choice, however, becomes important if S is executed in parallel with U. In general, the problem is that we cannot assert anything as a precondition for a statement since this precondition may be disturbed by other programs.

In order to cope with this problem, we define the operational semantics in two steps. In the first step we define for a program which sequence of atomic actions can arise from executing the program. An atomic action is *skip*, an assignment or a test. A sequence of atomic actions is called a trace (see also [19, 34]). In the second step, we define the set of state sequences associated with a set of traces. In contrast with chapter five, we now have to distinguish between a program and its meaning. We extend our notation with

$$\begin{array}{rl} \mathit{Synt} & = \text{The syntactical category which represents the programs that can be} \\ & \quad \text{written in Dijkstra's language, extended with parallel composition.} \\ \mathit{Traces} & = \text{The set of finite and infinite sequences over the atomic statements.} \\ & \quad \text{A test } B \text{ is denoted by } ?B. \text{ The empty trace is denoted by } \epsilon. \end{array}$$

Our two steps correspond to two functions.

$$\begin{array}{l} tr : \mathit{Synt} \to \mathcal{P}(\mathit{Traces}), \text{ gives for a program the set of traces of that program.} \\ s : \mathcal{P}(\mathit{Traces}) \to \mathit{Prog}, \text{ gives for a set of traces the set of state sequences associated} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{with that set.} \end{array}$$

The function s turns out to be quite trivial.

$$\begin{array}{rcl} s.\{\mathit{skip}\} & = & \{x : x \in X : xx\} \\ s.\{y := e\} & = & \{x : x \in X : x(x[y/e.x])\} \\ s.\{?B\} & = & B^{set} \\ s.\{uv\} & = & (s.u);(s.v) \\ s.A & = & \bigcup(t : t \in A : s.\{t\}),\ \text{for } A \subseteq \mathit{Traces} \end{array}$$

The important part is the function tr although that one is not very complicated either. We do maintain in a trace the information from which program the atomic actions originate. We might make this clear by subscripting every atomic action with the name of the program from which it stems but this seems pointless. Instead we provide the binary operator "in" for an atomic action in a trace and a program. Using this, we introduce the projection operator $\uparrow$. For $u \in \mathit{Traces}$

and $S \in Synt$, $u \uparrow S$ is the trace obtained from u by removing all atomic actions from u that are not in S, and maintaining the order of the remaining atomic actions.

Now we turn to defining tr for the constructs in our language. The constructs that already represent atomic actions are straightforward and so is sequential composition. We do not bother to make an explicit distinction between, for instance, an assignment viewed as an element of *Synt* or viewed as a trace.

$$\begin{aligned} tr.(y := e) &= \{y := e\} \\ tr.skip &= \{skip\} \\ tr.(S; U) &= (tr.S)(tr.U) \\ tr.abort &= \{(skip)^\infty\} \end{aligned}$$

In the alternative construct and in the repetition tests are introduced.

$$\begin{aligned} tr.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}) &= \bigcup(i :: (?B_i)tr.S_i) \cup \{(?\forall(i :: \neg B_i))(skip)^\infty\} \\ tr.(\textbf{do } B \rightarrow S \textbf{ od}) &= \bigcup(n : 0 \leq n : ((?B)tr.S)^n(?\neg B)skip) \cup ((?B)tr.S)^\infty \end{aligned}$$

Until now we did not change anything compared to chapter five, i.e., we can show that the operational semantics of chapter five appears as the composition of the functions tr and s. The advantage of the current definition is that we are able to include parallelism.

$$tr.(S \parallel U) = \{u : u \uparrow S \in tr.S \wedge u \uparrow U \in tr.U : u\}$$

(This definition is similar to the weaving operator, introduced in [34].) The traces of $S \parallel U$ are obtained by merging the traces of S and U.

Parallel composition makes it fairly impossible to reason about properties of programs in a compositional way like, for instance, has been done in chapters six and seven. It also forces us to remain in the domain of the operational semantics when proving facts about programs (like the requirements or the theorems in the previous chapters). There are now a number of possibilities. Firstly, we may, for all our reasoning about programs, rely on the operational semantics. This is very unattractive, even more than before since, due to the interleaving, every parallel program has an essential non-deterministic behavior. Secondly, we may try to find properties so strong that they allow us to conclude something about a parallel composition from the components of this composition. An example of such a property is p *unless* q. If p *unless* q is a property of program S then, if p becomes *true* during S it will remain *true* until q holds. If however, we define p *unless* q in the way described in chapter five, we have the same problem with parallel composition as we have with the other properties. Therefore we need a stronger property, more like the definition of properties in UNITY. Just as an example, we give it here.

$$\begin{aligned} &unless.skip.p.q && = true \\ &unless.abort.p.q && = true \\ &unless.(y := e).p.q && = [p \wedge \neg q \Rightarrow p^y_e \vee q^y_e] \\ &unless.(S; U).p.q && = unless.S.p.q \wedge unless.U.p.q \\ &unless.(\textbf{if } [](i :: B_i \rightarrow S_i) \textbf{ fi}).p.q && = \forall(i :: unless.S_i.p.q) \\ &unless.(\textbf{do } B \rightarrow S \textbf{ od}).p.q && = unless.S.p.q \\ &unless.(S \parallel U).p.q && = unless.S.p.q \wedge unless.U.p.q \end{aligned}$$

Notice that *unless.S.p.q* is a boolean, not a predicate. It expresses that p *unless* q is a property of every assignment in the program, regardless of their preconditions. Because of this, p *unless* q is a property of the whole program. Using properties like this, we may be able, for instance, to *prove* the rules of Gries-Owicki theory ([31]) rather than to have them as axioms.
Remark. The definition of *unless* differs from the UNITY definition in one important respect. In UNITY, the evaluation of a guard together with an assignment is considered to be an atomic action. We separated those two when we introduced parallelism.

A third possibility, and the one we think is most rewarding, is to restrict the class of programs. We may, for instance, restrict our attention to synchronization and communication primitives like semaphores ([10]) or synchronous communication on channels ([17]). We describe and prove the properties of such a primitive in the operational semantics. We then use these properties to reason about our programs at a higher level.

Bibliography

[1] de Bakker, J.W., Mathematical theory of program correctness, Prentice-Hall International, Englewood Cliffs New Jersey, 1980.

[2] Back, R.J.R., Refinement calculus, part II: parallel and reactive programs, report ser. A, no. 93, Åbo Akademi, Finland, 1989.

[3] Back R.J.R., Sere, K., Stepwise refinement of action systems, In: Mathematics of program construction (J.L.A. van de Snepscheut (ed)), Springer Verlag, LNCS 375, Heidelberg, 1989, pp. 115-138.

[4] Birkhoff, G., Lattice theory (3rd edition), American Mathematical Society, Providence, 1971.

[5] Chandy, K.M., Misra, J., Distributed computations on graphs: Shortest path algorithms, Communications of the ACM, vol. 25, no. 11, 1982, pp. 833-837.

[6] Chandy, K.M., Misra, J., Parallel programming, a foundation, Addison-Wesley publishing company, Reading, 1988.

[7] Dally, W.J., A VLSI architecture for concurrent data structures, Ph.D thesis, Computer Science, California Institute of Technology, report 5209-TR-86, 1986.

[8] Dijkstra, E.W., A discipline of programming, Prentice-Hall International, Englewood Cliffs New Jersey, 1976.

[9] Dijkstra, E.W., Scholten, C.S., Predicate calculus and program semantics, Springer-Verlag, New York, 1990.

[10] Dijkstra, E.W., Over seinpalen, EWD 74, Internal report.

[11] van Gasteren, A.J.M., On the shape of mathematical arguments, Ph.D. thesis, Computer Science, Eindhoven Technical University, The Netherlands, 1988.

[12] Gries, D., The science of programming, Springer-Verlag, New York 1981.

[13] Gratzer, G.A., General lattice theory, Academic Press, New York, 1978.

[14] Gratzer, G.A., Lattice theory; first concepts and distributive lattices, Freeman, San Francisco, 1971.

[15] Günther, K.D., Prevention of deadlocks in packet-switched data transport systems, IEEE Transactions on Communications, vol. C-29, no. 4, April 1981, pp. 512-524.

[16] Hilbers, P.A.J., Mappings of algorithms on processor networks, Ph.D. thesis, Computer Science, Groningen University, The Netherlands, 1989.

[17] Hoare, C.A.R., Communicating sequential processes, Communications of the ACM, vol 21 (8), pp. 666-677.

[18] Hoare, C.A.R., Communicating sequential processes, Prentice-Hall International, London, 1985.

[19] Hoare, C.A.R., Some properties of predicate transformers, Journal of the ACM, vol. 25, no. 3, July 1978, pp. 461-480.

[20] Hu, T.C., Combinatorial algorithms, Addison-Wesley Publishing Company, Reading, 1982.

[21] Jansen, J.M. and Sijstermans, F.W., A parallel implementation of the knapsack algorithm, Philips Res. Lab. Eindhoven, Doc. no. 152, February 1987.

[22] Kleinrock, L., Queuing Systems, vol. 2, Wiley, New York, 1976.

[23] Knapp, E., A predicate transformer for progress, IPL 33, (1989/1990), pp. 323-330.

[24] Kuiper, R., An operational semantics for bounded nondeterminism equivalent to a denotational one, In: Algorithmic Languages, de Bakker/van Vliet (eds), IFIP, North Holland, 1981, pp. 373-398.

[25] Manna, Z., Pnueli, A., How to cook a temporal proof system for your pet language, Proceedings POPL, Austin, ACM, 1983, pp. 141-153.

[26] Martin, A.J., The probe: an addition to communication primitives, IPL 20 (1985), April 1985, pp. 125-130.

[27] Martin, J.L., Mueller-Wichards, D.M., Supercomputer performance evaluation: status and directions, The Journal of Supercomputing 1, 1987, pp. 87-104.

[28] May, D., Shepherd, R., Keane, C., Communicating process architectures: Transputers and Occam, INMOS ltd., april 1986.

[29] Misra, J., Specifications of concurrently accessed data structures, In: Mathematics of program construction (J.L.A. van de Snepscheut (ed)), Springer-Verlag, LNCS 375, Heidelberg, 1989, pp. 91-114.

[30] Morris, J.M., Temporal predicate transformers and fair termination, Acta Informatica, 27(1990), pp. 287-313.

[31] Owicki, S., Gries, D., An axiomatic proof theory for parallel programs I, Acta Informatica, vol. 6, no. 1, 1976, pp. 678-686.

[32] Owicki, S., Lamport, L., Proving liveness properties of concurrent programs, ACM TOPLAS, vol. 4, no. 3, July 1982, pp. 455-495.

[33] Seitz, C.L., Concurrent VLSI architectures, IEEE Transactions on Computers, vol. C-33, no. 12, 1984, pp. 1247-1265.

[34] van de Snepscheut, J.L.A., Trace theory and VLSI design, Ph.D thesis, LNCS 200, Berlin, 1985.

[35] Vornberger, O., Load balancing in a network of Transputers, Proceedings of the 2nd international workshop on distributed algorithms 1987, LCNS 312, Springer-Verlag, March 1988, pp. 116-126.

[36] Warshall, S., A theorem on Boolean matrices, Journal of the ACM, 1962, pp. 1-2.

Index